



Le portage du systeme LE_LISP. Mode d'emploi

M. Devin

► To cite this version:

M. Devin. Le portage du systeme LE_LISP. Mode d'emploi. RT-0050, INRIA. 1985, pp.59. inria-00070108

HAL Id: inria-00070108

<https://inria.hal.science/inria-00070108>

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



CENTRE DE ROCQUENCOURT

Rapports Techniques

N° 50

LE PORTAGE DU SYSTÈME LE LISP MODE D'EMPLOI

Matthieu DEVIN

Mars 1985

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél : (1) 39 63 55 11

Le Portage du Système LE_LISP

Mode d'emploi

Matthieu Devin

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

Centre de Mathématiques Appliquées
Ecole des Mines de Paris
Sophia-Antipolis
06370 Valbonne
France

Rapport interne du projet VLSI
Mars 1985

Résumé : Ce manuel explique comment installer le système *LE_LISP* développé par Jérôme Chailloux à l'INRIA. Ce système comporte l'interprète, le compilateur et l'éditeur de texte d'un dialecte du langage LISP.

Abstract : This notice explains how to build the portable *LE_LISP* interpreter, devised by Jerome Chailloux of INRIA. It also covers the building of the compiler and an Emacs-like text editor.

Le système *LE_LISP* est en grande partie écrit dans le langage machine de la machine LLM3. Porter le système nécessite l'implémentation de ce langage. La méthode la plus efficace est la description, sous forme de *macros d'assemblage* de chaque instruction du langage. Nous expliquons ici comment implémenter LLM3, puis comment construire l'interprète *LE_LISP*. Nous expliquons ensuite comment construire le compilateur et comment gérer le terminal de l'utilisateur, afin d'avoir un système complet, muni d'un éditeur plein-écran.



CHAPITRE 1

Présentation

Nous esquissons d'abord la démarche à suivre. Nous précisons ensuite le matériel (fichiers) et les outils (programmes) nécessaires. Nous décrivons ensuite le langage dans lequel est écrit l'interprète, en donnant des conseils pour l'implémenter. Nous expliquons, enfin, comment mettre au point les premières versions de l'interprète, et comment étendre le système avec le compilateur et l'éditeur de texte.

Il est évidemment recommandé de connaître le langage LISP pour pouvoir l'implémenter en un temps raisonnable (de 1 à 4 mois); la mise-au-point en sera hautement accélérée.

1.1 Démarche à suivre

Vous disposez de 14 fichiers sources, constituant l'interprète LE_LISP écrit dans le langage machine LLM3. Porter cet interprète sur votre machine nécessite le portage de la machine LLM3. La meilleure méthode, du point de vue de l'efficacité, est de la décrire sous forme de *macros d'assemblage* : chaque instruction LLM3 est traduite en une ou plusieurs instructions de la machine cible. Vous devez donc traduire puis assembler les fichiers, pour ensuite les lier avec un lanceur et obtenir finalement l'interprète. Cette manière de faire impose d'avoir un macro-assembleur, ou bien un macro-expandeur qui traduira les fichiers LLM3, avant de les assembler. Dans la section *matériel* nous vous expliquons comment vous procurer un macro-expandeur.

On distingue 5 degrés de sophistication du système LE_LISP :

- Le *système nu* : C'est le résultat de l'expansion et de l'assemblage des fichiers LLM3, augmenté d'un toplevel minimum écrit en LISP. C'est déjà un interprète LISP complet de plus de 400 fonctions.
- Le *système standard* : C'est le système nu augmenté des sources LISP dits du *système standard*. Ceux-ci ajoutent un certain nombre de douceurs à l'interprète nu (sharp-macro, backquote-macro, etc.), et en particulier les outils de mise-au-point : paragrapheur, pisteuse etc.
- Le *système compilé* : C'est le système standard, augmenté du compilateur. Le compilateur est, bien sûr, écrit en LISP, et se compilera lui-même. Le portage du compilateur nécessite l'écriture, en LISP, d'un chargeur spécifique à votre machine. Voir au chapitre 5.
- Le *système avec éditeur* : C'est le système standard, augmenté des sources LISP de l'éditeur Pepe. C'est un éditeur plein-écran particulièrement étudié pour l'édition des fonctions LISP. Cet éditeur étant écrit en LISP, il est extrêmement facile de l'étendre.
- Le *système complet* : Il est constitué du système standard, du compilateur et de l'éditeur. Cela fait de LE_LISP un système très confortable, dont le seul

inconvenient peut être sa taille (environ 800 k-octets, ceci variant légèrement selon les machines).

Nous donnons, ci-dessous, un tableau des opérations à effectuer, en ordre chronologique. Les opérations du groupe I, conduisent au système nu puis au système standard. Ensuite on peut soit suivre le groupe II pour obtenir le système compilé, soit suivre le groupe III pour obtenir le système avec éditeur. Le système complet est obtenu en effectuant toutes les opérations.

Nous indiquons entre parenthèses les numéros des chapitres dans lesquels sont décrites ces opérations.

Groupe I

- 1 - Ecrire, éventuellement, un macro-expandeur (1).
- 2 - Etudier le langage LLM3 (2).
- 3 - Décrire chacune des instructions LLM3 à l'aide de macros, dans le langage d'assemblage de votre machine.
- 4 - Ecrire les routines nécessaires à certaines des instructions LLM3 : la bibliothèque d'exécution.
- 5 - Ecrire un lanceur (3).
- 6 - Expandre et assembler chacun des fichiers sources LLM3.
- 7 - Compiler/Assembler la bibliothèque.
- 8 - Compiler/Assembler le lanceur.
- 9 - Lier le tout en un unique programme : lelisp. Le tester avec les fichiers de test. Vous disposez maintenant d'un système LISP nu (4).
- 10 - Sous le système nu, charger les sources LISP, de l'environnement standard. Vous disposez maintenant d'un système LISP standard. Faites-en une image mémoire.

Groupe II

- 1 - Ecrire, en LISP, un chargeur pour le compilateur (5).
- 2 - Sous le système LISP nu, charger les sources LISP du compilateur. Compiler l'environnement standard, ce qui inclut évidemment le compilateur lui-même. Vous disposez alors d'un système-compilé. Faites-en une image mémoire.

Groupe III

- 1 - Installer éventuellement un fichier de description de terminaux. LE LISP dispose d'un "compilateur" capable de transformer une description d'un terminal dans la syntaxe de termcap (standard UNIX) en une série de fonctions LISP. Une copie d'un fichier termcap est donc éventuellement suffisante.
- 2 - Charger les sources LISP de l'éditeur Pepe. Vous disposez alors d'un système avec éditeur. Faites-en une image mémoire.

1.2 Matériel nécessaire

1.2.1 Fichiers sources

- Sources LLM3 : 14 fichiers nommés :

llinit.llm3, toperr.llm3, physio.llm3, gc.llm3, read.llm3, print.llm3, eval.llm3, cntrl.llm3, carcdr.llm3, fntstd.llm3, extend.llm3, number.llm3, string.llm3, blsht.llm3.

- Sources LISP : 25 fichiers répartis en cinq catégories :

Le toplevel minimum, 4 fichiers :

startup.ll, macroch.ll, defs.ll, virtty.ll.

L'environnement standard, 7 fichiers :

sort.ll, sysmac.ll, trace.ll, pretty.ll, array.ll, sort.ll, callex.ll.

Le compilateur, et le chargeur, 2 fichiers :

llcp.ll, lapXXX.ll.

L'éditeur Pepe, 2 fichiers :

pepe.ll, pephelp.ll.

Les tests, démonstrations et statistiques, 10 fichiers :

testfn.ll, testcp.ll, testdata.ll, testfib.ll, testblt.ll, testlap.ll, hanoi.ll, whanoi.ll, vdt.ll, statllm3.ll.

1.2.2 Outils

- Un macro-expandeur.

C'est le programme qui va traduire, à partir de vos définitions, les fichiers sources LLM3 en fichiers sources assembleur.

Si vous disposez d'un macro-assembleur, c'est lui qui fera ce travail. Il se peut cependant qu'un simple macro-assembleur ne soit pas suffisant : il peut être nécessaire de transformer la syntaxe des arguments des macro-instructions ou de réaliser différentes expansions selon les types des arguments. Vous devez alors, soit écrire vous-mêmes un macro-expandeur, soit en utiliser un déjà existant. Nous pouvons vous en fournir plusieurs. Voyez le chapitre 2.

- Un assembleur.

Pour assembler les fichiers sources LLM3, après leur expansion. Il servira aussi à assembler le lanceur et la bibliothèque.

- Un éditeur de liens.

1.2.3 Documentation

Vous avez aussi besoin d'une bonne documentation sur votre machine et sur votre système d'exploitation, d'une documentation LE LISP correspondant à la version que vous installez (ici version 15 du 31 Décembre 1984), et... de ce manuel pour vous guider.

1.3 Temps de portage

Selon le temps dont vous disposez vous installerez une version plus ou moins complète de l'interprète. Voici une idée des temps nécessaires à chacune des opérations élémentaires constituant le minimum nécessaire au portage :

Ecriture d'un macro-expandeur : 3-9 jours, selon le programmeur.

Ecriture des définitions de macros : 2-4 jours, selon la machine cible.

Ecriture de la bibliothèque : 1 jour.

Ecriture du lanceur : 1 jour.

Expansion totale, assemblage sans erreur : 4 jours.

Edition de liens et premiers galops : 2-8 jours, selon la connaissance de LISP.

Ces temps s'entendent pour une personne connaissant bien le système sur lequel elle travaille, et travaillant à temps plein au portage. Selon la machine cible, le système d'exploitation et le programmeur, ces temps peuvent être multipliés par 2 ou 3. Comptez donc un bon mois pour avoir un système LISP nu. Si vous pouvez utiliser le macro-expandeur écrit en LISP et cross-expanser vos sources LLM3 hors-site, vous pouvez gagner la moitié du temps.

Pour obtenir un système avec compilateur, il vous faudra 1 semaine supplémentaire, si vous connaissez LISP, beaucoup plus sinon.

Pour obtenir un système avec éditeur-vidéo, fonctionnant sur vos terminaux, comptez une demi-journée.

CHAPITRE 2

L'expansion

Nous expliquons ici ce que doit être un macro-expandeur et nous en proposons de trois types.

2.1 Qu'est un macro-expandeur?

Un macro-expandeur est un programme qui traduit, au moyen d'un ensemble de définitions, les fichiers LLM3 en fichiers assembleurs.

Les sources LLM3 sont constitués des 14 fichiers suivants :

llinit.llm3, toperr.llm3, physio.llm3, gc.llm3, read.llm3, print.llm3, eval.llm3, cntrl.llm3, carcdr.llm3, fntstd.llm3, extend.llm3, number.llm3, string.llm3, blisht.llm3.

Chaque fichier constitue un module indépendant, pouvant être assemblé séparément.

Pour expander ces fichiers LLM3 vous devez écrire un fichier contenant les définitions, dans le formalisme de votre expandeur, de chaque instruction LLM3.

La démarche générale se fait donc selon le schéma :

```

Fichier LLM3 -----> Fichier Assembleur -----> Code Binaire
    expandeur              assembleur
  
```

2.2 Nos macro-expandeurs

Nous décrivons ici deux expandeurs écrits en C et un expandeur écrit en LISP. Le premier est très compact, effectuant une substitution simple des arguments. Le second permet d'expander différemment les instructions selon le type de leurs arguments. Le dernier permet une expansion très accordée à la machine but. C'est l'expandeur en LISP qui est maintenant utilisé pour réaliser les versions LE LISP VAX, MC68000, RIDGE32, HB68/Multics et BellMac32. Il doit être utilisé pour tout système bootstrappé. Les macros expandeurs en C ne seront utilisés que lorsque le bootstrap est impossible.

2.2.1 Le premier expandeur en C

Une définition de macro à la forme :

```
FOO      MACRO
        <corps de la macro>
        ENDM
```

Où <corps de la macro> est constitué d'un nombre quelconque de lignes d'assembleur.

Lors de l'expansion, chaque occurrence de l'instruction FOO est remplacée par <corps de la macro>. Dans le corps certaines chaînes de caractères ont des significations spéciales. Ces chaînes sont :

\1, \2, \3, \4, \5, \6, \7, \8, \9 et \@

Les chaînes \i, i valant de 1 à 9, sont remplacées par le i^{ème} argument de l'appel de la macro.

La chaîne \@ est remplacée par une valeur différente pour chaque macro-expansion. Ceci permet la génération d'étiquettes locales à chaque appel de macro.

Par exemple, avec les définitions (pour Perkin Elmer 32) :

```
BTEQ      MACRO
          cmp      \1,\2
          be       \3
          ENDM
```

```
BTSYMB     MACRO
          cmp      \1,BSYMB
          bl       \@
          cmp      \1,ESYMB
          bl       \2
          equ      *
          ENDM
```

Les instructions :

```
BTEQ      A1,.$12,FOOBAR
BTSYMB     CAR(A3),LOOP
```

Sont expansées en :

```
          cmp      A1,.$12
          be       FOOBAR

          cmp      CAR(A3),BSYMB
          bl       1100
          cmp      CAR(A3),ESYMB
          bl       LOOP
1100      equ      *
```

De plus, dans le macro-expandeur, certaines fonctions permettent de modifier la syntaxe des arguments des instructions LLM3. Ceci est utile, par exemple, si votre assembleur ne note pas les nombres hexadécimaux de la même manière que LLM3. Vous devez donc redéfinir ces fonctions pour convenir à vos notations.

2.2.2 Le second expandeur en C

Le second est un sur-ensemble du premier. Dans les corps de macros il reconnaît en plus les chaînes :

\t1, \t2, \t3, \t4, \t5, \t6, \t7, \t8, \t9, \a, \ta.

\a est expansé en la liste des arguments de l'appel, quelque soit leur nombre.

\ti, i allant de 1 à 9, est le type du i^{ème} argument de l'instruction. Ce type est donné sous forme d'une lettre : r pour registre, i pour immédiat, o pour objet, et u si l'argument n'est pas fourni. Voyez la description des types dans la section 2.

\ta est une chaîne de caractères formée des types de tous les arguments fournis à la macro, mis côte à côte.

Par exemple avec la définition suivante (sous ensemble du MOV PerkinElmer) :

```
MOV      MACRO
          ifz      \t2-r
          lr       \2,\1
          else
          l        \2,\1
          endif
        ENDM
Les instructions LLM3 :
MOV      A1,A2
MOV      #20,A2
sont expansés en :
          ifz      r-r
          lr       A2,A1
          else
          l        A2,A1
          endif
          ifz      i-r
          lr       A2,##$20
          else
          l        A2,##$20
          endif
```

Ici dans la macro MOV on utilise l'assemblage conditionnel de l'assembleur cible pour générer des parties de code différentes selon le type des arguments.

De plus, le corps de la macro peut comporter des structures de la forme :

```
          case      <argument>
<val1>:    ..
          ...
<val2>:    ...
          ...
def:       ...
          ...
        endcase
```

Où, <argument> est n'importe quoi pouvant prendre les valeurs <val1>, <val2>, etc. On expande la structure case-endcase en la partie de code correspondant à la valeur de <argument>. Si aucun des <vali> ne convient, on expande la partie de code correspondant à la valeur 'def', si elle est présente. Si aucune valeur ne correspond, l'expansion de cette expression ne produit rien.

Par exemple la définition :

```

MOV      MACRO
case      \ta
rr:      lr      \2.\1
rp:      st      \1.\2
endcase
ENDM

produit pour :
MOV      A1,A2
MOV      A4,IBASE
les instructions :
lr      A2,A1
st      A4,IBASE

```

2.2.3 Le macro-expandeur LE_LISP

Nous proposons enfin un macro-expandeur écrit en LE_LISP. Pour pouvoir l'utiliser vous devez disposer d'un interprète ou d'un compilateur sur une autre machine et d'une bonne liaison entre cette machine et la votre (ligne asynchrone 9600 bauds, bande magnétique). En effet la taille totale des sources assembleurs résultat de l'expansion des sources LLM3 atteint couramment 300K octets. Le temps de transfert sur une ligne asynchrone à 1200 bauds est donc d'au moins 40 minutes (le transfert prend en fait couramment plus d'une heure).

l'expansion se fait alors selon le schéma :

- 1) Machine distante :
LLM3 -----> Assembleur
- 2) Transmission des fichiers Assembleur sur votre machine
- 3) Machine locale :
Assembleur -----> Binaire

Pour l'expansion du VAX nous utilisons en fait l'interprète LE_LISP de la version antérieure pour expander une nouvelle version. Le tout est de toujours pouvoir disposer d'un interprète... Cet expandeur en LISP est si pratique que nous l'utilisons dorénavant pour toutes les nouvelles machines.

L'expandeur en LISP permet de générer un code plus optimisé que les autres expandeurs, son emploi doit donc se généraliser, au moins pour le suivi des versions après un premier portage, lorsqu'il est possible d'utiliser l'expandeur sur le site même de compilation.

2.2.4 Fonctionnement

L'expandeur LE_LISP fonctionne en deux temps.

Les sources LLM3 sont d'abord traduit en format LAP : chaque ligne source devient une Liste contenant l'étiquette éventuelle (()) par défaut), le code opération et les arguments.

Les arguments sont transformés selon le tableau :

Type	Notation LLM3	Représentation LISP
<i>Registres</i>	A1	A1
<i>Immédiats décimaux</i>	#23	(# 23)
<i>Immédiats hexadécimaux</i>	#\$FFFC	(# \$FFFC)

Immédiats symboliques
Immédiats adresses pures
Immédiats adresses impures
Immédiats LISP
Chaînes
Mémoires directes
Mémoires indirectes

#MARK
 @RETOUR
 %CHEAP
 .T
 "On y va.."
 IBASE
 CAR(A1)

(# MARK)
 (@ RETOUR)
 (% CHEAP)
 .T
 "On y va.."
 IBASE
 (CAR A1)

Soit par exemple :

```

LOOP      MOV      A1,A2      (LOOP MOV A1 A2)
          PUSH     #1         (( PUSH (# 1))
          CNBEQ    A2,CAR(A1),FIN  (( CNBEQ A2 (CAR A1) FIN)

```

La deuxième phase de l'expansion consiste à appeler une fonction correspondant à chaque code-op avec pour arguments les arguments de l'instruction. Cette fonction doit s'occuper d'imprimer le code machine correspondant à l'appel de la macro-instruction.

Ces fonctions sont aisément écrites grâce à un jeu de macro-caractères approprié. Comme toute la puissance de LISP est disponible dans chaque macro on peut aisément accorder le code produit aux différents cas de figure (type des arguments, arguments constants).

Voici par exemple la définition des instructions arithmétiques pour l'implémentation de LLM3 sur MC 68000.

```

(defns PLUS (arithmetique 'ADD 1 2 3))
(defns DIFF (arithmetique 'SUB 1 2 3))

```

```

(de arithmetique (op arg1 arg2 label)
  (optim-op op arg1 arg2)           ; fait le calcul dans D0
  (ifn label
    ; pas de test de débordement
    [" MOVE.L D0," arg2]           ; range le résultat du calcul
    ; il faut tester le débordement
    ; (mais APRES avoir range le resultat)
    [" SVS D1"]                   ; mémorise le débordement éventuel
    [" MOVE.L D0," arg2]           ; range le résultat du calcul
    [" TST.B D1"]                  ; teste le débordement mémorisé
    [" BNE " label]))              ; le calcul a débordé!

```

```

(de optim-op (op arg1 arg2)
  ; calcule arg1 op arg2 dans D0 (op est ADD ou SUB)
  [" MOVE.L " arg2 ".D0"]
  (cond ((short-immédiat? arg1)
    [" " op "Q.W " arg1 ".D0"]) ; calcul immédiat court
    ((memory? arg1)
    [" " op ".W " arg1 "+2.D0"]) ; calcul mémoire
    (t
    [" " op ".W " arg1 ".D0"]))) ; cas standard

```

CHAPITRE 3

Le langage LLM3

Nous décrivons ici le langage LLM3 et donnons des conseils d'implémentation. La référence complète pour le jeu d'instruction est constitué par l'article "La Machine LLM3" de Jérôme Chailloux.

3.1 Quelques unités

Un POINTEUR est une quantité de la taille d'une adresse. Par exemple 32 bits sur un VAX11/780 ou un MC68000.

Un OCTET est une quantité de 8 bits.

3.2 Le langage d'assemblage

Le langage LLM3, ressemble beaucoup à un assembleur symbolique classique. Une ligne a la forme :

<étiquette> <instruction> <opérandes> <commentaires>

Seule la partie <instruction> est obligatoire, toutes les autres sont optionnelles. Le champ commentaire commence par le caractère point-virgule (;).

La partie <opérandes> est composée d'un ou plusieurs opérandes, séparés par des virgules. Les opérandes déterminent le mode d'adressage de l'instruction. Ils sont de quatre types :

- Registre rapide
- Objet direct (adresse mémoire)
- Objet indirect (adressage indirect indexé)
- Valeur immédiate (constante numérique, adresse, constante LISP)

Un opérande de type *registre* est noté A1, A2, A3 ou A4. Il représente l'un des quatre registres de la machine LLM3. Sa valeur est le contenu du registre spécifié.

exemples :

```
PUSH    A1
MOV     A4,A2
```

Les opérandes de type *mémoire directe* permettent d'accéder aux variables globales de LE_LISP. L'opérande est une adresse symbolique. La valeur de l'opérande est le contenu de cette adresse.

exemples :

```
MOV      BCONS, A1
PUSH     OBASE
```

Les opérandes de type *mémoire indirecte* permettent d'accéder aux champs des objets de types CONS et SYMB, ainsi qu'au champ VAL des objets de types NUMB, FLOAT, VECT et STRG. On utilise pour ce faire un adressage indirect indexé.

Les opérandes de ce type ont la forme :

XXX(Ai)

où Ai est l'un des quatre registres, et XXX un déplacement permettant l'accès au champ. La valeur de XXX(Ai) est le contenu de l'adresse mémoire (Ai)+XXX, où (Ai) représente le contenu du registre Ai. Le déplacement XXX est donné sous forme symbolique : CAR, CDR, CVAL, etc.

exemples :

```
PUSH      CDR(A1)
MOV        CAR(A2), FVAL(A4)
MOVN1L     VAL(A4)
```

Les opérandes de type *immédiat* servent à représenter des valeurs numériques, des adresses, ou bien des objets LISP.

Les *immédiats numériques* commencent par le caractère #, suivi d'un nombre décimal, d'un nombre hexadécimal précédé du caractère \$, ou d'un symbole. La valeur de l'opérande est la valeur du nombre, ou du symbole.

exemples :

```
PUSH      #12
MOV        #$FF32, A1
SFTYPE     #SUBRO, A4
```

Les *immédiats de type adresse* sont de deux types : adresses pures et adresses impures. Les adresses pures sont des adresses de la zone code, les adresses impures sont des adresses de la zone données. Les *immédiats adresses impures* commencent par le caractère % les *immédiats adresse pures* par le caractère @. La valeur de l'opérande est l'adresse en question.

exemples :

```
IMPURE
CHEAP     ADR      0
PURE
RETOUR    LABEL
PUSH      @RETOUR
HGOBJ     %CHEAP, A1
```

Les *immédiats objets lisp* permettent de manipuler les atomes LISP en LLM3. Ils sont constitués d'un point (.) suivi d'un nom de symbole précédemment défini par MAKCST ou MAKFNT. Leur valeur est l'adresse de l'*atome lisp* créé par l'instruction MAKCST ou MAKFNT.

exemples :

```
MAKFNT     PRINT.5, "print"
MOV        .PRINT, A1 ; A1 <- l'adresse du symbole PRINT
```

3.3 Description de la machine

3.3.1 Les registres

La pseudo machine LLM3, comporte 4 registres, notés A1, A2, A3 et A4. Ces quatre registres sont de taille suffisante pour contenir des POINTEURS. Ils sont, de plus, totalement interchangeables.

3.3.2 La pile

La machine LLM3 est dotée d'une pile, d'une taille d'au moins 4000 POINTEURS. Elle peut contenir indifféremment des adresses ou des valeurs. Elle sert de pile de contrôle et de pile de données. En LLM3 il n'y a pas de registre *pointeur de pile* accessible, l'accès à la pile s'effectue au moyen d'instructions spécialisées.

3.3.3 Objets manipulés

LLM3 manipule principalement des POINTEURS. Il y a cependant un groupe d'instructions spécialisées pour manipuler les caractères (OCTET), et un autre groupe pour les nombres qui, nous le verrons, sont des pointeurs particuliers.

3.3.4 Les types LISP

Il y a six types d'objets LISP, stockés dans six zones disjointes de la mémoire :

nombres entiers (zone NUMB), nombres flottants (FLOAT), vecteurs (VECT), chaînes de caractères (STRG), symboles (SYMB) et cellules de listes (CONS).

Le stockage des nombres entiers est cependant légèrement différent de celui des autres objets, nous y reviendrons plus loin.

Chaque zone de la mémoire est décrite par deux pointeurs :

- un pointeur de début de zone : BNUMB, BFLOAT ... BCONS.
- un pointeur de premier objet libre : CNUMB, CFLOAT ... CCONS.

Le pointeur de premier objet lisp est positionné par les systèmes qui réalisent une initialisation statique des zones (uniquement le PC d'IBM aujourd'hui).

Les zones sont gérées par l'interprète au moyen d'instructions spécialisées permettant d'avancer dans les zones (NXNUMB, NXFLOAT ...) de marquer les objets utilisés (STMARK, TCMARK ...) et d'allouer de nouveaux objets (CONS, FADD, ...).

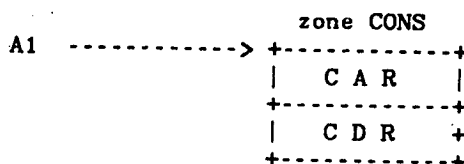
Vous devez, lors du lancement initialiser les pointeurs Bxxx et Cxxx de chaque zone. Nous discuterons de la taille des zones lors de l'écriture du lanceur. Voir le chapitre 2.

Nous décrivons maintenant plus en détail la structure des objets de chaque zone et les instructions LLM3 y correspondant.

3.3.4.1 Le type CONS

Les objets de la zone CONS, sont des paires de POINTEURS. Ils ont donc pour taille 2 POINTEURS. Le premier champ de la paire s'appelle le CAR, le second le CDR.

Une cellule de liste (ou CONS), peut se schématiser par :



En LLM3, CAR et CDR sont des symboles indiquant le déplacement nécessaire pour accéder aux deux champs d'un cons. Vous devez, dans vos définitions, leur donner les valeurs requises par votre implémentation.

par exemple :

```

CAR      EQU      0
CDR      EQU      4
  
```

sur une machine à pointeurs 32 bits adressant l'octet (VAX, MC68000).

3.3.5 Initialisation de la zone

Lors de l'initialisation des zones (à chaque lancement de LE_LISP) et après chaque nettoyage (GC), les cellules libres sont chaînées par leur champ CDR, les champs CAR sont initialisés à NIL et le pointeur de liste libre (FREEL) est mis sur la première cellule libre. Le CDR de la dernière cellule de la liste libre est NIL, indiquant la fin de la liste.

La construction d'une nouvelle cellule (instructions CONS, XCONS, NCONS) se fait donc en prenant la première cellule de la liste libre et en remplissant les champs. L'instruction CONS, par exemple, peut se définir en LLM3 de la manière suivante :

Définition de CONS A1,A2 ; A2 <- (A1 . A2)

```

      MOV      FREEL,A3      ; le premier doublet libre
      BFNIL    A3,OK        ; c'est bien une cellule
      CALL     GCCONS        ; plus de doublets, on appelle la GC
      MOV      FREEL,A3      ; le premier doublet récupéré
OK     MOV      CDR(A3),FREEL ; avance dans la liste libre
      MOV      A1,CAR(A3)    ; positionne le CAR
      MOV      A2,CDR(A3)    ; positionne le CDR
      MOV      A3,A2        ; rend le doublet dans A2
  
```

Notons que cette définition n'est absolument pas utilisable car elle modifie le registre A3; l'algorithme est néanmoins correct.

3.3.5.1 Le type SYMBOLE

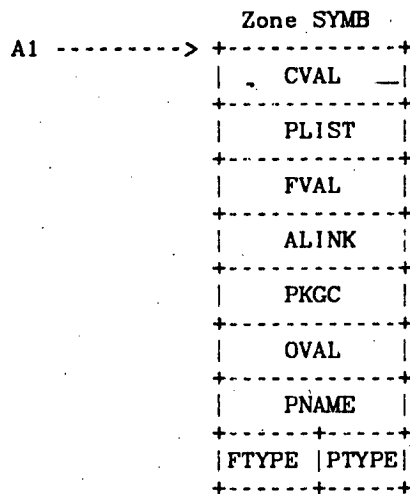
Les objets de la zone SYMBOLE, sont constitués de :

- 7 champs contenant un POINTEUR : CVAL, PLIST, FVAL, ALINK, PKGC, OVAL, PNAME.
- 2 champs contenant un OCTET : ftype, ptype.

Les champs contenant un POINTEUR servent à stocker les différentes propriétés des symboles. On y accède par un adressage indirect indexé, au moyen de la valeur des symboles correspondants.

On accède aux autres champs par des instructions spéciales.

Les objets de la zone SYMB peuvent être représentés par le schéma :



Les objets de la zone SYMBOLE ont donc pour taille 7POINTEURS+2OCTET.

3.3.6 Initialisation

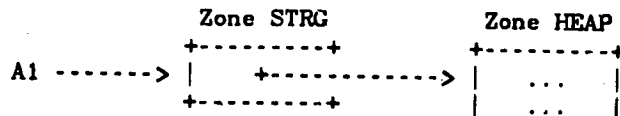
Au lancement du système (dans linit), et après chaque nettoyage (GC), les symboles libres sont chaînés par leur champ ALINK. Le champ ALINK du dernier symbole libre contient la constante NIL.

3.3.6.1 Le type chaîne de caractère (STRG)

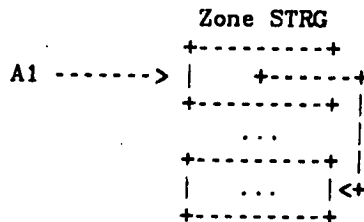
Les objets de la zone chaîne sont des pointeurs dans la zone HEAP. Ils disposent d'un seul champ, VAL, qui contient donc une adresse du HEAP. Ce champ VAL sert aussi à chaîner les chaînes libres (à l'initialisation et après chaque nettoyage). Le champ VAL de la dernière chaîne libre pointe sur la constante NIL.

Les Chaînes se représentent selon le schéma :

CHAÎNE ALLOUÉE



CHAÎNE LIBRE

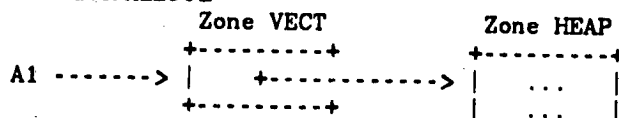


3.3.6.2 Le type vecteur (VECT)

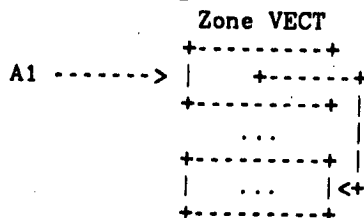
Les objets de la zone vecteur sont aussi des pointeurs dans la zone HEAP. Ils disposent d'un seul champ, VAL, qui contient donc une adresse du HEAP. Ce champ VAL sert aussi à chaîner les vecteurs libres (à l'initialisation et après chaque nettoyage). Le champ VAL du dernier vecteur libre pointe sur la constante NIL.

Les Vecteurs se représentent selon le schéma :

VECTEUR ALLOUÉ



VECTEUR LIBRE



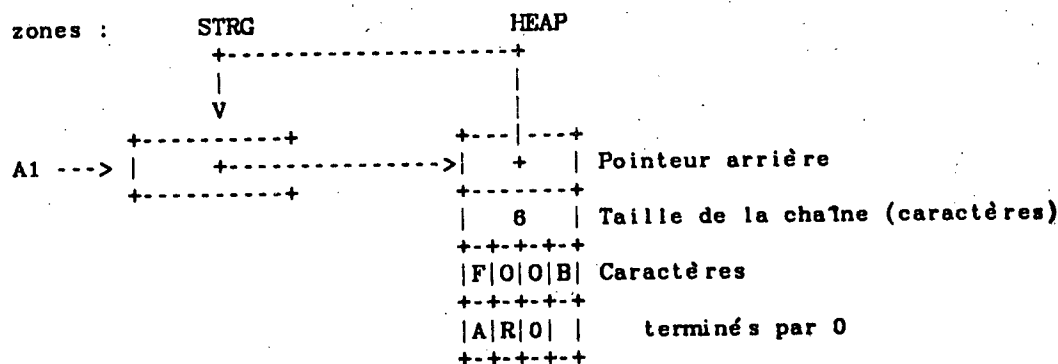
3.3.6.3 Le tas (HEAP)

Le tas (ou HEAP) est la zone mémoire où l'on stocke le contenu des chaînes et des vecteurs. C'est la seule zone contenant des objets de taille variable.

Les objets du HEAP ne sont pas des objets LISP au même titre que les atomes, les chaînes, ou les autres. Les objets du HEAP sont manipulés indirectement lors de la manipulation des chaînes de caractères et des vecteurs, et ce par des instructions spécialisées. LLM3 manipule cependant des pointeurs directs dans le HEAP lors de la recompaction (après nettoyage). Dans ce cas les pointeurs sont toujours des opérandes mémoire et jamais des registres (Les registres LLM3 contiennent toujours des objets LISP).

Les objets de la zone HEAP sont des objets de taille variable. Ils sont décrits par leur taille, un pointeur arrière sur l'objet qu'ils représentent (STRG ou VECT) et la suite de caractères (STRG) ou de pointeurs (VECT) constituant cet objet.

- A1 est un pointeur dans la zone STRG
- L'objet de cette zone est un pointeur dans la zone HEAP
- L'objet de la zone HEAP est formé
 - d'un pointeur arrière sur l'objet de la zone STRG
 - d'un mot contenant la longueur de la chaîne (ici 6 caractères)
 - des caractères proprement dits, terminés par un zéro (pour UNIX)



```

zones :      VECT                                HEAP
              +-----+                         +-----+
              |         |                         |         |
              v         |                         |         |
A1 ---> +-----+ +-----+ Pointeur arrière
        |   +-----+ |   +-----+
        +-----+ +-----+
                               +-----+
                               |   4   | Taille du vecteur (pointeurs)
                               +-----+
                               +-+-+-+-+
                               | +-----> Symbole A
                               +-+-+-+-+
                               | +-----> Symbole B
                               +-+-+-+-+
                               | +-----> Symbole C
                               +-+-+-+-+
                               | +-----> Symbole D
                               +-+-+-+-+
                               +-+-+-+-+

```

Les objets du HEAP sont manipulés indirectement par les instructions sur les vecteurs et les chaînes de caractères : HPMOVX, HBMOVX, HPXMOV, HBXMOV, HGSIZE, HSSIZE, HGOBJ, HSOBJ, NHXP, NXHB. Toutes ces instructions travaillent sur des objets de type STRG ou VECT et permettent d'accéder au HEAP de manière interne (cachée à LLM3).

Imprimé le Vendredi 22 Mars 1985

Examinons par exemple l'instruction HPXMOV qui permet de lire le n-ième élément d'un vecteur :

HPXMOV A1,A2,A3

A1 est un entier indiquant le numéro de l'élément que l'on désire connaître.

A2 est un vecteur (pointeur dans la zone VECT).

A3 doit recevoir le résultat.

Cette instruction se compile (sur MC68000) en :

```

MOVE.L      (A2),A0      ; pointeur dans le HEAP
ADD.L       #8,A0        ; déplacement vers les données
MOVE.L      A1,D0        ; l'index
LSL.L       #2,D0        ; multiplié par 4 (adresse d'octets)
MOVE.L      (A0,D0.L),A3 ; le résultat

```

3.3.9 Manipulation directe

Les objets du HEAP sont manipulés directement par les deux instructions CHBLT et HBLT.

CHBLT (Compare Heap-address and Branch Less Than) est une comparaison d'adresses de la zone HEAP. Cette instruction est utilisée pour déterminer si il y a suffisamment de place dans le HEAP pour allouer une nouvelle chaîne ou un nouveau vecteur.

HBLT (Heap BLiT) permet de déplacer des zones entières du HEAP (à des fins de recompaction de la mémoire). Cette instruction prend 3 opérandes qui sont des adresses de la zone HEAP :

HBLT AA,BB,CC

a pour effet de recopier à l'adresse CC toute l'information stockée de l'adresse AA (incluse) à l'adresse BB (excluse).

3.3.9.1 Les nombres flottants (FLOAT)

Les nombres flottants sont des pointeurs dans la zone FLOAT. Ces objets disposent d'un champ VAL qui permet de chaîner les flottants libres (à l'initialisation et après chaque nettoyage).

A chaque pointeur est associée une valeur flottante, dont la précision dépend de la machine. Cette valeur peut être indifféremment stockée :

- à la suite du champ VAL dans la zone FLOAT
- dans la zone HEAP en utilisant la pointeur VAL (Cf STRG et VECT)
- ailleurs (dans un autre segment sur HB68/Multics)

Toutes les instructions LLM3 rendant un résultat flottant doivent créer un nouvel objet dans la zone FLOAT. Pour cela il faut utiliser la variable FFLOAT qui est un POINTEUR sur le prochain objet libre de cette zone. Les objets libres sont chaînés par leur champ VAL, le champ VAL du dernier objet est la constante numérique 0.

L'allocation d'un nouveau flottant se fait donc de la manière suivante :

```

          CNBNE      FFLOAT,#0,OK      ; le prochain flottant est-il libre ?
          CALL       GCFLOAT           ; non, appel du GC
OK        MOV       FFLOAT,A1         ; On prend le pointeur dans A1
          MOV       VAL(A1),FFLOAT     ; avance dans la zone FLOAT
<ici on met dans le champ valeur de A1, le flottant à allouer>

```

N'oubliez pas que chaque instruction flottante doit générer cette partie de code pour stocker son résultat.

3.3.9.2 Les nombres entiers (NUMB)

Le système LELISP manipule des petits nombres entiers (sur 16 bits). Ceci permet sur la plupart des machines de ne pas utiliser de zone pour stocker les nombres mais plutôt de les considérer comme de valeurs immédiates. Ceci est possible car toutes les zones de données commencent en général au delà de l'adresse \$FFFF (65535), à cause de la place occupée par le code LLM3 compilé. Ceci rend l'arithmétique entière extrêmement efficace (pas d'allocation ni de nettoyage nécessaire).

Une telle implémentation n'est cependant pas possible sur toutes les machines, par exemple sur le 8086 qui utilise des pointeurs 16 bits. Il existe donc une zone NUMB, pour permettre une implémentation des nombres entiers en mémoire.

Si l'on utilise une telle implémentation il est nécessaire que toutes les opérations arithmétiques créent un nouvel objet dans cette zone pour le résultat.

3.3.9.3 la zone CODE

La zone CODE contient le code produit par le compilateur. Elle est décrite par les deux variables globales LLM3 : BCODE et ECODE.

Cette zone mémoire est entièrement gérée en LISP, il n'est donc pas utile de la décrire ici.

3.4 Espace mémoire

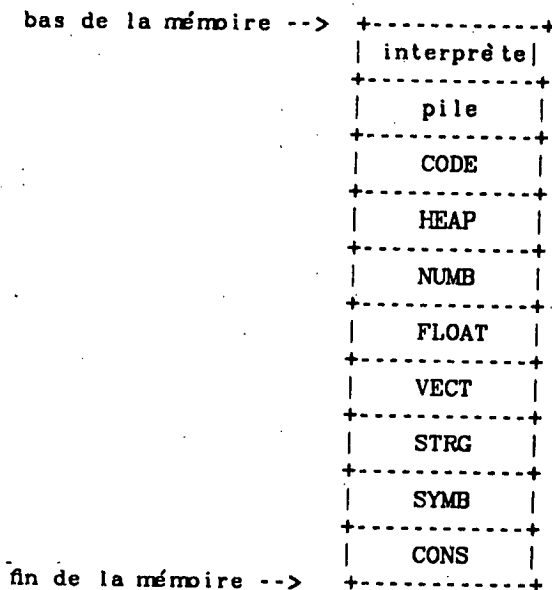
Nous expliquons ici la manière standard de découper l'espace mémoire adressable par l'interprète.

Cet espace est divisé en 10 zones : interprète (LLM3), pile, CODE, HEAP, NUMB, FLOAT, VECT, STRG, SYMB et CONS.

L'utilisation de la zone NUMB (petits entiers en mémoire) est facultative. Cette zone est nécessaire pour les systèmes disposant de pointeurs 16 bits, comme l'IBM PC. On peut aussi se passer de la zone CODE si l'on n'installe pas le compilateur (pour les petits systèmes comme l'IBM PC, le Macintosh), et de la zone FLOAT, si l'on n'implémente pas de nombres flottants.

Notez cependant que les zones non utilisées doivent pouvoir contenir au moins un objet du type concerné.

L'organisation classique de ces zones est la suivante :



Cette manière de faire permet d'avoir un test de type optimisé pour les CONS, car une seule comparaison d'adresse suffit.

De plus elle rend facile l'assimilation des objets des zones pile, interprète, CODE et HEAP, au type NUMB. Cette assimilation est nécessaire dans le récupérateur lors du marquage de la pile.

3.5 Implémentation

3.5.1 Préliminaires

Pour implémenter LLM3, vous devez :

- Choisir un emploi pour chacun des registres de votre machine
- Donner des valeurs aux symboles définissant l'accès aux champs des types CONS et SYMB. Définir le symbole VAL pour les objets de types FLOAT, STRG et VECT.
- Décrire chaque instruction sous forme de macro, dans votre langage machine.
- Ecrire les routines nécessaires à votre implémentation.

3.5.2 Allocation des registres

Du point de vue de l'efficacité il est que les quatre registres (A1, A2, A3, A4) soient des registres de votre machine.

Si votre machine ne dispose pas de pile cablée, il est impératif de consacrer un registre au rôle de *pointeur de pile*. Le *pointeur de pile* n'est jamais référencé

directement en LLM3, mais vous en aurez besoin pour implémenter des instructions comme PUSH, POP, CALL etc.

Il est aussi recommandé de stocker l'adresse du symbole || (que l'on appelle NIL en LLM3, mais qui n'est pas le symbole LISP NIL) dans un registre. Ceci accélère les tests BxNIL, et les tests de types BxSYMB, car cet atome est le premier objet de la zone SYMB. Pour permettre ceci, || est manipulé par des instructions spéciales LLM3.

S'il vous reste encore des registres disponibles, vous pouvez les utiliser pour contenir des copies des variables globales LLM3 contenant les adresses des autres limites de zones, de manière à accélérer les tests de types.

En priorité affectez donc les registres aux usages suivants :

- registres LLM3 (A1, A2, A3, A4) —
- pointeur de pile, si vous en avez besoin.
- constante || (qui est aussi NIL et BSYMB)
- début de la zone CONS, POINTEUR BCONS.
- début de la zone CHAINE, POINTEUR BSTRG.
- autres limites de zones (BFLOAT, BVECT, BVAR)
- bloc de contrôle courant dans la pile, POINTEUR PBIND.
- pointeur de liste libre
- autres variables globales à discrétion.

3.5.3 Initialisation de vos registres

Vous devez initialiser les registres contenant les limites de zones et le pointeur de pile lors du lancement du système. Cette opération est par exemple effectuée par votre lanceur.

3.5.4 Accès aux champs des objets LISP.

Vous devez donner des valeurs aux symboles suivants :

- pour les CONS : CAR, CDR.
- pour les SYMBOLES : CVAL, PLIST, FVAL, ALINK, PKGC, OVAL, PNAME.

Pour les autres objets vous devez définir le symbole VAL.

Par exemple pour une machine à mots de 8 bits où le POINTEUR vaut 32 bits, en suivant les schémas du chapitre 3.3.4.2. les valeurs des symboles sont :

CAR	EQU	0
CDR	EQU	4
CVAL	EQU	0
PLIST	EQU	4
FVAL	EQU	8
ALINK	EQU	12
PKGC	EQU	16
OVAL	EQU	20
PNAME	EQU	28
VAL	EQU	0

3.6 Description des instructions

Les instructions de la machine LLM3 sont décrites dans l'article de Jérôme Chailloux *La machine LLM3*.

3.7 La bibliothèque (runtime)

Les instructions LLM3 étant particulièrement simples, la bibliothèque contient en général uniquement les routines nécessaires aux entrées/sorties, et aux instructions comme COMLINE, GETENVRN, etc.

Nous proposons dans l'annexe A une bibliothèque écrite en C réalisant ces instructions. Nous vous conseillons de vous en inspirer pour écrire votre bibliothèque. Notez bien que ces routines gagnent peu de choses à être écrites en assembleur car leur efficacité n'est généralement pas primordiale.

3.8 Sortie de LE_LISP

En sortie, l'interprète exécute un branchement à l'adresse *LL_EXIT*. Cette adresse est déclarée XREFP dans le fichier *llinit.llm3*, elle ne correspond à aucun code. Le saut à cette adresse est prévu pour vous permettre de sortir proprement de l'interprète. Vous devez écrire un petit programme à l'adresse *LL_EXIT* pour sortir de LE_LISP proprement.

CHAPITRE 4

Le lanceur

Le lanceur est le programme qui initialise votre système avant de rentrer sous LE_LISP. Nous détaillons ici ce qui doit être initialisé, et nous vous proposons un lanceur écrit en C.

Le minimum indispensable consiste en :

- l'initialisation des variables globales LE_LISP.
- le passage des arguments de la commande d'appel "lelisp"
- l'allocation de la mémoire.
- l'initialisation des variables globales de la bibliothèque.
- l'initialisation de la ligne de communication (en mode caractère sans echo).

Une fois les initialisations faites vous devez rentrer sous LE_LISP au point *LLINIT*. C'est une étiquette du fichier *llinit.llm3*, déclarée XDEFB. Il suffit de faire un branchement en ce point.

4.1 Les variables globales LE_LISP

Les variables globales de LE_LISP sont des variables déclarées XDEFB en LLM3 que le lanceur *doit* initialiser avant de se brancher en *LLINIT*.

4.1.1 Définition des zones

Vous devez initialiser toutes les variables LLM3 indiquant les limites des zones.

Il y a deux variables pour chaque zone indiquant l'adresse du début de la zone et l'adresse du premier objet libre de la zone; elles ont pour noms B<zone> et C<zone>. En général la variable C<zone> contient la même valeur que la variable B<zone>. C<zone> recevra une autre valeur si vous décidez de réaliser une initialisation statique du système (aujourd'hui seul le PC d'IBM est concerné par ce type d'initialisation).

Les zones CODE et HEAP sont décrites par des variables indiquant l'adresse du début et la fin des zones : B<zone> et E<zone>.

les variables à initialiser sont donc :

BCODE, ECODE : zone CODE.
 BHEAP, EHEAP : zone HEAP.
 BNUMB, CNUMB : éventuelle zone NUMB.
 BFLOAT, CFLOAT : zone FLOAT.
 BSTRG, CSTRG : zone STRG.
 BVECT, CVECT : zone VECT.
 BSYMB, CSYMB : zone SYMB.

BCONS, CCONS : zone CONS.

Notez bien que :

- 1 - LLM3 ne connaît pas les adresse de fin de zones (pour les zones "LISP"). ceci n'est pas nécessaire, grâce aux instructions NX<zone> qui font un test de débordement de zone.
- 2 - Les tailles des zones sont fixes une fois le system lancé.

4.1.2 Les limites de la pile

Bien que ne gérant la pile que par des instructions spécialisées, LLM3 a besoin d'en connaître les limites. Elles sont stockées dans trois variables globales : *BSTACK*, *ESTACK*, *MSTACK*.

BSTACK est l'adresse de début de la pile.

ESTACK est l'adresse de fin de pile.

MSTACK est une adresse comprise entre *BSTACK* et *ESTACK*, représentant la valeur du pointeur de pile lorsque qu'il ne reste plus que 64 *POINTEUR* disponibles avant débordement.

Vous devez, avant de lancer LE_LISP, donner les valeur que vous avez choisies pour ces variables.

4.1.3 Autres variables

Il faut positionner la variable *NBSYST* qui indique le numéro de votre système.

Les numéros des systèmes ont été définis comme suit :

1 = VERSADOS,	2 = VME,	3 = MicroMega,	4 = APOLLO
5 = SM90,	6 = PE32OS,	7 = PE32UNIX,	8 = VAXUNIX
9 = VAXVMS,	10= multics,	11= METHEUS,	12= UNIVERSE68
13= MCPM86,	14= PCDSO,	15= LISA,	16= VAXIS3,
17= MAC,	18= SPS9,	19= BELLMAC,	20= VM37OUTS

4.2 Les options de lancement

La syntaxe standard d'un appel (au niveau commande système) à LE_LISP est :
lelisp [-s] [-r file] [file] [scons]

Les arguments entre crochets sont optionnels. Ils ont la signification suivante :

-s pas d'impression de la bannière
-r file restaurer l'image mémoire de nom file
file lire le fichier initial file
scons est un nombre qui précise la taille de la zone cons.

Chaque option (sauf *scons*) correspond à une variable globale LLM3 qui doit être positionnée par le lanceur au lancement.

Ces variables sont *FILIT*, *FILIN*, *FILIZ* et *LLBAN*.

FILIZ vaut 0 ou 1. Si FILIZ vaut 1, il faut lire un fichier au lancement, si FILIZ vaut 0 il faut charger une image mémoire. FILIN indique le nom de l'objet.

FILIN est une chaîne de caractères, c'est à dire un pointeur sur des caractères (non nécessairement terminée par le caractère NULL). Elle représente, selon la valeur de FILIZ, le nom du fichier initial à lire, ou de l'image mémoire à restaurer.

FILIT est une variable de la taille d'un POINTEUR contenant la longueur de la chaîne FILIN. Si FILIT vaut 0 il n'y a ni fichier à lire, ni image mémoire à restaurer.

LLBAN vaut 0 ou 1. Si LLBAN vaut 0, LE_LISP imprime la bannière au lancement, si LLBAN vaut 1 le lancement est silencieux. Ceci est utile pour construire des sous-systèmes.

Dans le fichier *llinit.llm3* ces variables sont déclarées par les lignes :

	XDEFI	FILIN	
	XDEFI	FILIZ	
	XDEFI	FILIT	
FILIZ	ADR	0	: taille du nom du fichier
FILIT	ADR	0	: type (0 = fichier, <>0 = image memoire)
FILIN	ADR	0	: nom
	XDEFI	LLBAN	
LLBAN	ADR	0	: type de la banniere (0 normal, 1 muet)
	XDEFI	NBSYST	
NBSYST	ADR	0	: Numero du systeme.

4.2.1 Taille de la zone CONS, et des autres zones.

La taille de la zone CONS doit pouvoir être changée à chaque appel du système, les autres tailles sont constantes. L'argument *scons* passé à la commande *lelisp* permet de préciser la taille voulue. Pour cet argument, l'unité est le 8k cellule de liste. Par exemple un appel de LE_LISP avec un argument 4 donne une zone CONS de taille $4 \cdot 8 \cdot 1024$ cellule de liste soit $4 \cdot 8 \cdot 1024 \cdot 2$ POINTEUR. Si l'argument numérique n'est pas fourni une valeur par défaut sera utilisée. Nous recommandons une valeur par défaut de 3 pour un système d'un méga-octet de mémoire centrale, de 5 sur un système plus ample.

Si utilisez un système LE_LISP *sans compilateur* et que vous disposez de 512 K-octet de mémoire, une bonne répartition est :

taille CODE	0
taille HEAP	80k OCTET
taille FLOAT	2k objets = 2k POINTEUR
taille VECT	1k objets = 1k POINTEUR
taille STRG	3k objets = 3k POINTEUR
taille SYMB	2k objets = 16k POINTEUR
taille CONS	<i>scons</i> * 8k objets = <i>scons</i> * 16k POINTEUR

Pour un système *avec compilateur*, dans un espace d'un méga-octet nous vous recommandons la répartition :

taille CODE	200k OCTET
taille HEAP	200k OCTET

taille FLOAT	2k objets = 2k POINTEUR
taille VECT	3k objets = 1k POINTEUR
taille STRG	5k objets = 3k POINTEUR
taille SYMB	3k objets = 16k POINTEUR
taille CONS	scons*8k objets = scons * 16k POINTEUR

Pour calculer la taille globale d'un appel à LE_LISP, sachez que le code de l'interprète fait 80K OCTET sur VAX et 110k OCTET sur 68000.

4.2.2 Note Importante

En raison des choix d'implémentation aucun zone ne soit être vide (sigh!). Si vous n'utilisez pas une zone donnée (NUMB, FLOAT par exemple) il est nécessaire de lui donner une taille permettant de manipuler 1 (un) objet au moins.

4.3 Allocation de la mémoire

Vous devez, une fois les tailles connues, allouer une taille mémoire suffisante à votre programme, pour pouvoir contenir toutes les données.

Si l'on demande plus de mémoire que le système ne peut en donner, on doit terminer immédiatement en erreur.

4.4 initialisation de la bibliothèque

Si les routines que vous utilisez pour les instructions LLM3 ont besoin d'une initialisation, faites la dans le lanceur.

4.5 Initialisation de la ligne de communication

La communication avec le terminal doit se faire caractère par caractère et sans écho. Vous aurez donc peut-être besoin de changer les paramètres de la ligne de communication avec l'utilisateur appelant LE_LISP. Ceci doit être fait dans le lanceur.

Il ne faut pas oublier, à la sortie de LE_LISP, de restaurer l'état initial de la ligne de communication. Notez bien que ceci doit aussi être fait pour l'instruction COMLINE.

Si vous devez stocker quelquepart des données représentant l'état initial de la ligne de communication, faites en sorte que les instructions COREST et CORSAV ne les modifient pas, de graves chosent peuvent en découler.

4.6 divers

Vous aurez peut-être aussi besoin de

- préparer le contrôle des interruptions
- etc.

Toutes ces opérations doivent être faites dans le lanceur.

4.7 Un exemple de lanceur en C

Nous vous proposons dans l'annexe B un lanceur complet écrit en C. Nous vous conseillons de vous en inspirer pour écrire le votre.

CHAPITRE 5

Le premier interprete

Nous détaillons ici l'assemblage final du premier interprete.

Rappelons que vous devez d'abord expander tous les sources LLM3, puis les assembler. Vous devez ensuite assembler votre bibliothèque, et votre lanceur, et enfin lier le tout en un programme executable : LE_LISP. Il est alors temps d'essayer de faire tourner ce programme qui constitue l'interprete nu.

5.1 Le lancement

Pour le premier lancement de LE_LISP, nous vous conseillons de ne pas demander de fichier initial. Ceci vous permet d'essayer de faire tourner l'interprete sans la gestion de fichier, partie que vous pourrez rajouter plus tard sans problème.

Donnez donc aux variables d'initialisation les valeurs suivantes :

```
FILIZ 0
FILIT 0
FILIN N'importe quoi (car FILIT vaut 0)
LLBAN 0
```

Au lancement de LE_LISP, une bannière doit s'imprimer. Elle a la forme :

```
***** LE_LISP (by INRIA) version 15 (31/Decembre/84) [vaxunix]
```

où le nom du système utilisé (à la place de [vaxunix]) correspond à la valeur de la variable globale NBSYST.

Ensuite un point d'interrogation apparaît en début de ligne, c'est le caractère d'invite LE_LISP.

Vous êtes alors sous le contrôle de la fonction TOPLEVEL qui va sans cesse lire des expressions, les évaluer et imprimer leur valeur.

Si tout se passe ainsi, vous pouvez passer directement au test de l'interprete (section 4.4).

Dans la plupart des cas, cependant, c'est autre chose qui arrive, souvent une erreur fatale, un trap hardware, une erreur du bus, etc.

Nous étudions ici différents cas de figure et montrons comment essayer de faire tourner LE_LISP correctement. Expliquons d'abord la séquence d'initialisation.

5.2 Initialisation

L'initialisation procède en plusieurs phases :

- Variables globales, limites de zones
- Modules LLM3

Pour suivre pas à pas l'initialisation de l'interprète, donnez, dans les fichiers LLM3, la valeur 1 au symbole DEBUG. Ceci entraîne l'impression d'un message informatif, *avant* chaque phase de l'initialisation. Nous passons ici les différents messages en revue, et détaillons les instructions LLM3 utilisées dans chaque phase.

5.2.1 Messages GC_XXXX

Ils correspondent à l'initialisation des zones lisp : NUMB, FLOAT, VECT, STRG, SYMB et CONS. Pour chaque zone on utilise l'instruction NX<zone>, et les modes d'adressage correspondant aux champs des objets de cette zone (VAL, CAR, CDR et ALINK). On utilise aussi l'instruction MOVNIL.

5.2.2 INL_NIL

Initialisation de la table de hachage : la table de hachage étant un vecteur LISP on utilise à ce niveau toutes les instructions des vecteurs : NXHP, CHBLT, HPMOVX, HSSIZE, HSOBJ, etc.

Ce cap est difficile à passer car il met beaucoup de choses en jeu. Notamment si les instructions NXHP ou CHBLT sont mal codées vous risquez d'appeler le garbage-collector!

Construction de la chaîne "" et du vecteur #[] : on utilise encore toutes les instructions du HEAP, des vecteurs et des chaînes.

Construction de l'atome NIL, puis de l'atome UNDEF (MAKCST).

Construction des premières fonctions (MAKFNT).

5.2.3 INL_TOP

Initialisation des constantes et fonctions contenues dans le module toperr.llm3 (MAKCST, MAKFNT).

5.2.4 INL_GC

Fonctions du module gc.llm3 (MAKFNT).

5.2.5 INL_PIO

Fonctions du module physio.llm3 (MAKFNT).

Tampon de lecture (instructions sur les chaînes).

5.2.6 INL_REA

Fonctions et constantes du module read.llm3 (MAKFNT, MAKCST).

Table des caractères (HBMVX).

5.2.7 INL_PRI

Fonctions et constantes du module print.llm3 (MAKFNT, MAKCST).

Tampon d'impression (instructions sur les chaînes).

5.2.8 INL_EVA

Fonctions et constantes du module eval.llm3 (MAKFNT, MAKCST).

Point d'entrée pour le compilateur (ADRHL, CONS).

5.2.9 INL_CTL

Fonctions et constantes du module cntrl.llm3 (MAKFNT, MAKCST).

5.2.10 INL_CAD

Fonctions et constantes du module carcdr.llm3 (MAKFNT, MAKCST).

5.2.11 INL_STD

Fonctions et constantes du module fntsd.llm3 (MAKFNT, MAKCST).

5.2.12 INL_NBS

Fonctions et constantes du module nbstrs.llm3 (MAKFNT, MAKCST).

5.2.13 INL_EXT

Fonctions et constantes du module extend.llm3 (MAKFNT, MAKCST).

5.2.14 INL_BLL

Fonctions et constantes du module bllsht.llm3 (MAKFNT, MAKCST).

5.3 En cas de difficultés

Nous étudions ici différents cas où l'initialisation de LE_LISP ne se passe pas comme elle devrait. Nous donnons quelques idées de remèdes.

5.3.1 Aucun message d'initialisation n'est imprimé

Vérifiez que lors de l'assemblage le symbole DEBUG était bien à 1.

L'instruction TTYMSG est mal codée; vérifiez la à la main.

L'initialisation du module linit.llm3 défaille, vérifiez les instructions concernées, en mode pas-à-pas sous votre débogueur symbolique favori.

5.3.2 La bannière ne s'imprime pas

L'impression de la bannière se fait *après* l'initialisation des modules LLM3. Réassemblez vos fichiers avec le symbole DEBUG à 1. Si l'un des messages d'initialisation ne s'imprime pas c'est que le module précédent est défaillant. Vérifiez alors les définitions des instructions utilisées dans ce module.

Si la bannière ne s'imprime toujours pas, vérifiez les instructions : TTYSTRG, HBMOVX, et TTYCRLF.

5.3.3 Pas d'écho des caractères lus

Trois causes principales :

- Le canal de communication n'est pas en mode caractère sans echo.
- l'instruction TTYIN est mal codée, vérifiez-la.
- la table des caractères est mal initialisée, examinez son contenu à la main : c'est une chaîne LISP qui se trouve dans la variable globale TABCH.

5.4 Chargement de l'environnement standard

Si l'interprète semble marcher correctement, on peut tester les fonctions d'entrée/sortie. Essayez donc de charger le fichier startup.ll.

Notez qu'à ce stade la mise-au-point est grandement accélérée car vous pouvez la faire en LISP.

Le fichier startup.ll contient le toplevel écrit en LISP, le système de trace de la pile, les macrocaractères #. Une fois ce fichier chargé vous êtes sous un LISP minimum.

Les extensions LISP du système (formatteur, pisteur, arrays, etc...) sont définies en autoloading et seront donc chargées au moment utile.

Nous conseillons de lire (et de comprendre) ce fichier initial pour bien comprendre le toplevel LISP.

Le fichier startup.ll contient aussi la définition d'un certain nombre de variables

LISP indiquant les directories système standard. Vous devez editer ce fichier pour donner les valeurs correspondant à votre système.

Ce fichier contient aussi la définition de la fonction *load-std*, qui permet de charger l'environnement standard, l'éditeur et le compilateur.

Cette fonction demande 5 arguments ayant dans l'ordre la signification suivante :

- 1- création d'une image mémoire de nom: "<argument>.core" dans le directorie "#:system.core-directory". Si cet argument vaut () on ne crée pas d'image.
- 2- chargement de l'environnement minimum (t on charge, () on ne charge pas)
- 3- chargement de l'éditeur Pepe. (t on charge, () on ne charge pas).
- 4- chargement de l'environnement standard (t on charge, () on ne charge pas).
- 5- chargement du chargeur et du compilateur (sous format LAP) (t on charge, () on ne charge pas).

Par exemple l'appel :

(load-std 'lelisp t () ()) charge l'environnement minimum et en fait une image mémoire de nom *lelisp.core*.

Vous pouvez alors relancer ce système par la commande :
lelisp -r lelisp.core

Si vous avez des problèmes à ce niveau-là, essayez à la main en LISP les fonctions qui semblent incorrectes. N'hésitez pas à vous plonger dans les fichiers LLM3 pour découvrir leur code et comprendre quelles instructions sont mal définies.

5.5 Test de l'interprète

Lorsque, après un long labeur, l'interprète semble tourner correctement, il est temps de le tester à fond.

Les fichiers de test (testfn.ll testcp.ll) réalisent un test (pas encore assez complet à notre goût) automatique des fonctions du système standard lelisp.

Leur mise en œuvre est simple, il suffit de les charger (dans l'environnement standard) par la commande :
(libload <nom-du-fichier>)

Le test est alors automatique. Des messages informatifs sont imprimés au fur et à mesure des tests, et des messages très voyant sont imprimés en cas d'erreur. Vous devez alors vérifier à la main les formes incriminées pour corriger les instructions LLM3 défailantes.

Les tests se répartissent comme suit :

- testfn.ll : test des fonctions standard
- testcp.ll : test des mêmes fonctions après passage par le compilateur et le chargeur
- testlap.ll : test du chargeur.

CHAPITRE 6

Le Compilateur

Le compilateur **LE_LISP** est contenu dans le fichier **LISP : llcp.ll**.

Il utilise un chargeur, contenu dans le fichier : **lapXXX.ll**. **XXX** étant le nom de votre système.

Le compilateur est donné sous sa forme **LAP**. Pour pouvoir le charger il faut donc d'abord avoir écrit le chargeur.

Nous expliquons d'abord l'utilisation du compilateur, ensuite le rôle du chargeur et comment en écrire un pour votre système.

6.1 Qu'est le compilateur

Le compilateur est un programme **LISP** qui traduit des fonctions **LISP** en langage machine. Il procède en deux étapes :

- 1/ traduction du **LISP** en **LAP** (**LISP Assembly Program**).
- 2/ traduction du **LAP** en langage machine.

Le **LAP** est un langage intermédiaire, très proche de **LLM3**.

La première étape de traduction se fait par le compilateur. Le chargeur s'occupe de la seconde.

Pour le moment le **LAP** n'est pas exactement du **LLM3**. Il le sera sans doute un jour.

6.2 Comment compiler

La compilation **LE_LISP** se fait in situ : pour compiler une fonction vous devez d'abord appeler l'interprète puis, sous son contrôle, appeler le compilateur. La compilation est donc totalement interne à l'interprète **LE_LISP**.

Un appel au compilateur a la forme :
? (compile <function> <deep> <listLAP> <listHEXA>)

<function> est le nom de la fonction compiler, les autres arguments sont des flags, mis à *nil* s'ils ne sont pas fournis. On les positionne en leur donnant la valeur *t*. Positionnés ils ont les significations suivantes :

<deep> appel récursif du compilateur sur toutes les fonctions appelées par la fonction à compiler.

<listLAP> impression d'un listing du **LAP** généré.

<listHEXA> impression d'un listing hexadécimal du code fourni par le chargeur.

6.3 Ecriture du chargeur

Le chargeur est, bien sûr, écrit en LISP. Il utilise la fonction lisp MEMORY qui permet de lire/écrire directement des codes binaires dans la zone CODE.

La fonction MEMORY correspond exactement aux instructions LLM3 MEMSET et MEMGET. Vous pouvez implémenter la fonction MEMORY pour écrire des OCTETS, des MOTS (2 OCTETS) ou bien toute quantité qui sied à votre système.

Sur VAX, MEMORY permet de charger des OCTETS, sur 68000 on charge des MOTS, sur HB68 on charge des quantités 36 bits.

Nous vous fournissons plusieurs chargeurs, commentés, destinés à différentes machines :

lapvax.ll pour VAX11
lap68k.ll pour MC 68000
lape32.ll pour Perkin Elmer 32
lapmult.ll pour HB68

Ces deux chargeurs sont bâtis autour du même noyau : un module s'occupant de la résolution des étiquettes, et des références externes.

Ils diffèrent uniquement par l'ensemble des fonctions qui chargent en mémoire le code machine. Nous n'avons hélas pas de documentation sur le LAP aujourd'hui (Janvier 1985), et donc pas de liste détaillée des instructions que doit connaître le chargeur. La seule documentation disponible est constituée des 4 chargeurs que nous fournissons (Vax, MC68000, Perkin-Elmer, HB68).

Nous vous conseillons de vous inspirer de ces chargeurs (qui sont commentés) pour écrire le votre.

6.4 Test du Compilateur

Le fichier testlap.ll permet de tester le chargeur. Sa mise en oeuvre est similaire à celle du fichier testfn.ll.

Une fois que le chargeur est censé marcher vous pouvez essayer de charger le compilateur (qui est en format LAP).

Vous testerez alors le compilateur et le chargeur avec le fichier de tests testcp.ll, dont la mise en oeuvre est similaire à celle du fichier testfn.ll.

S'il s'avérait impossible de charger le compilateur sous format LAP (si le chargeur ne marche pas assez bien), il faut obtenir le compilateur en format source pour pouvoir faire des tests exhaustifs des instructions LAP.

CHAPITRE 7

Le terminal virtuel

Le système LE LISP dispose d'un certain nombre de fonctions permettant de réaliser des applications plein-écran. L'ensemble de ces fonctions constitue la description d'un *terminal virtuel*.

L'implémentation du terminal virtuel dépend évidemment du type de terminal qui est connecté au système. L'initialisation du terminal virtuel est réalisé par la fonction *initty* qui charge un fichier de définitions de fonctions correspondant au type de ce terminal.

A cet effet on garde dans le directory *#system.virtty-directory* des fichiers LISP de nom *<type>*. Il contenant les définitions des fonctions plein-écran pour les terminaux *<type>*. Si les terminaux que vous utilisez ne sont pas déjà décrits, prenez modèle sur les fichiers fournis et décrivez-les. Vous pouvez aussi utiliser le compilateur TERMCAP-VIRTTY fournit avec le système.

7.1. les variables globales

#tty:xmax est le nombre de colonnes du terminal moins une. Par exemple 79 sur un terminal 80 colonnes.

#tty:ymax est le nombre de lignes du terminal moins une. Par exemple 23 sur un terminal 24 lignes.

7.2 Les fonctions

Les fonctions sont toutes dans le package *#tty:<nom-du-terminal>*.

typrologue prépare le terminal à l'utilisation d'un éditeur plein écran.

tyepilogue prépare le terminal à l'utilisation normale de LISP (passage en mode "rouleau" par exemple).

tycursor (*x y*) positionne le curseur en colonne *x* ligne *y*. Le coin en haut à gauche de l'écran a pour coordonnées (0, 0).

tybeep fait sonner la clochette.

tycls efface tout l'écran.

tycleos efface l'écran à partir du curseur.

tycleol efface la ligne à partir du curseur.

tyinsch (*n*) insère le caractère *n* à la position du curseur.

tyinstr (*l*) insère la liste de caractères *l* à la position du curseur.

tydelch efface le caractère à la position du curseur.

tyinsln insère une ligne blanche à la position du curseur.

tydelln efface la ligne où est le curseur.

tyattrib (*i*) si *i* vaut *t* positionne l'attribut à la position du curseur, si *i* vaut *nil* annule l'attribut à la position du curseur. L'attribut peut être au choix: soulignage, vidéo inverse, demi-intensité etc.

Toutes ces fonctions retournent *()* si l'action demandée n'est pas réalisable par le terminal.

7.3 Initialisation du terminal virtuel

C'est la fonction LISP INITTY qui s'occupe de cette initialisation.

Voici son fonctionnement :

- Elle peut recevoir le nom du terminal de trois manières :
 - en argument
 - en évaluant (*getenv "TERM"*)
 - en évaluant (*system*)
- Elle cherche alors à charger le fichier *<nom-du-terminal>.ll* dans le directory *#:system:virtty-directory*
- Si ce fichier n'existe pas elle essaie de le créer à partir du fichier *termcap* (dont le path est dans la variable *#:system:termcap-file*).

L'initialisation peut donc se faire à la main à tout moment par un appel à *initty* avec pour argument, qui est évalué, le type du terminal.

Une bonne politique des systèmes utilisant le terminal virtuel est d'appeler la fonction INITTY après le save-core : le terminal sera initialisé à chaque appel du système.

Vous pouvez tester vos fichiers de description de terminaux avec le fichier de démonstration hanoi de la bibliothèque lisp. Évaluez simplement :

```
? (libload hanoi)
= hanoi
? (hanoi 5)
```

Annexes

Nous donnons ici les sources du lanceur et de la bibliothèque C que nous utilisons pour tout nos systèmes UNIX. Il est conseillé de s'en inspirer et même de les recopier pour gagner du temps dans l'implémentation du système.

7.4 Le lanceur C

Nous donnons ci dessous un listing du lanceur C que nous utilisons sur tous les systèmes UNIX.

```

.....

    Le Lanceur LE_LISP V15
    utilisable sur tous les UNIX.

    Matthieu Devin
    Jerome Chailloux

    Décembre 1984

.....

/*
   Les Unites avec lesquelles sont définies les zones
*/

#define PTR      (sizeof (char *))
#define KPTR     (PTR*1024)

/*
   Les tailles des zones (attention aux Unites!)
   -----
*/

#ifndef SSTACK
#define SSTACK 8                      /* en K objets de type pointeurs */
#endif
#ifndef SCODE
#define SCODE 200                     /* en K octets */
#endif
#ifndef SHEAP
#define SHEAP 140                     /* en K octets */
#endif
#ifndef SVECT
#define SVECT 8                      /* en K objets de type vecteur */
#endif
#ifndef SNUMB
#define SNUMB 1                      /* en K objets de type entiers */
#endif
#ifndef SFLOAT
#define SFLOAT 2                     /* en K objets de type flottant */
#endif

```

```

#ifndef SSTRG
#define SSTRG 5 /* en K objets de type chaine */
#endif
#ifndef SSYMB
#define SSYMB 3 /* en K objets de type symboles */
#endif
#ifndef SCONS
#define SCONS 3 /* en 8 K CONS */
#endif

/*
Choix du numero du systeme :
-----
1 = VERSADOS, 2 = VME, 3 = MicroMega, 4 = APOLLO
5 = SM90, 6 = PE320S, 7 = PE32UNIX, 8 = VAXUNIX
9 = VAXVMS, 10= multics, 11= METHEUS, 12= UNIVERSE68
13= MCPM88, 14= PCDOS, 15= LISA, 16= VAXIS3,
17= MAC, 18= SPS9, 19= BELLMAC, 20= VM370UTS,
21= PCS,

*/

#ifndef NBSYST
#define NBSYST 8
#endif

/*
Les autres parametres dependants du systeme
-----
*/

#ifndef FILEINI
#define FILEINI "/udd/lelisp/chailloux/ll/lib/startup.ll"
#endif
#ifndef LELISPBIN
#define LELISPBIN "/udd/lelisp/chailloux/ll/vax/lelispbin"
#endif

#define FILIT 0 /* 0 = fichier initial, 1 = core */
#define LLBAN 0 /* 0 = banniere, 1 = silence */

/* parametres du lanceur */

#define OK 16 /* signal de communication avec lelispgo */

/* les include */

#include <sys/types.h> /* pour tout le monde */
#include <signal.h> /* pour les signaux */
#include <stdio.h> /* pour les entrées sorties */
#include <errno.h> /* pour perror(), et le save-core BSD */
#include <a.out.h> /* pour getglobal, et le save-core BSD */

#ifdef BSD42 /* pour runtime */
#include <sys/time.h>
#include <sys/resource.h>
#else
#ifdef Perkin /* le Perkin n'a pas l'include sys/times.h */
struct tms {
long tms_utime;

```



```

    long    proc_system_time;
    long    child_user_time;
    long    child_system_time;
};
#else
#include <sys/times.h> /* pour les autres */
#endif
#endif

/* les point d'entree de LE_LISP LLM3 */
extern llistart(), llistdio();

/*
    Les variables globales LE_LISP.
    Elles sont toutes definies dans LLINIT.LLM3
    mais doivent etre chargees ici.
*/

/* la pile d'evaluation de LE_LISP */
extern char *bstack, *estack, *mstack;

/* les limites des zones des differents types LE_LISP */
extern char    *bcode, *ccode, *ecode,
                *bheap, *cheap, *eheap,
                *bnumb, *cnumb,
                *bfloat, *cfloat,
                *bvect, *cvect,
                *bstrg, *cstrg,
                *bsymb, *csymb,
                *bcons, *ccons, *econs;

/* le fichier initial et la ligne de commande */
extern int filiz, filit;
extern char *filin;
extern int liban;

/* le numero (type) du systeme LE_LISP */
extern int nbsyst;

/* Le flag controlant l'impression des erreurs système */
extern int **prtmgs; /* il est dans le save-core ! */
#define errreturn(M,V) { if(**prtmgs != 0) perror(M); return(V); }

/*
    Les variables du lanceur
    Elles ne sont pas sauvees par save-core
*/

/* tailles des zones */
int sstack, scode, sheap, svect, snumb, sfloat,sstrg, ssymb, scons;

/* bits invisibles */

```

```

int stbin;
char *btbin;

/* bits du GC */
int stbge;
char *btbge;

/* debut et fin de la memoire */
char *bmem;
char *emem;

/* File descriptor du pipe permettant de communiquer avec le lispge */
int pipilgo;

/* Environnement Shell initial (pour le restore-core BSD) */
char **envpini;

/*
  Points d'entree
*/

/* Pour les erreurs */
int out(), oupps(), usage();

/*
  References externes
*/

/* Pour les erreurs */
extern int codereturn(); /* dans l1stdio.c */

/* Allocation de la memoire */
extern char *sbrk();
#define RATE (char *) -1

/*
  Lancement du systeme
*/

main(argc, argv, envp)
int argc; char **argv, **envp; {
int n, i;
char *c;
int size;

#ifdef VAX
int selfcore = 0;
#endif

/*
  Trappe des signaux
*/

#ifdef DEBUG
for(i = 1; i <= 15; signal(i++, oupps));
#endif

#ifdef VAX
envpini = envp; /* sauvegarde de l'environnement Shell */
#endif

```

```

nlin = FILEINI; /* initialisation des valeurs par default */
nlit = FILIT;
liban = LLBAN;

sstack = SSTACK * KPTR;
scode = SCODE * 1024;
sheap = SHEAP * 1024;
svect = SVECT * KPTR;
snumb = SNUMB * KPTR + PTR; /* au moins 1 */
sfloat = SFLOAT * KPTR * 3 + PTR; /* au moins 1 */
sstrg = SSTRG * KPTR;
ssymb = SSYMB * KPTR * 8; /* symbole = 8 pointeurs */
scons = SCONS * KPTR * 16; /* c'est en 8K CONS */

nbsyst = NBSYST;

/*
  Decryptage des arguments

  Le premier argument est TOUJOURS le fd du pipe avec lelispgo.
*/

piplgo = (argc==1) ? 2 : atoi(argv[1]);

for(n = 2; n < argc; n++)
  if((*(argv+n) >= '0') && (*(argv+n) <= '9'))
    scons = 2*8*KPTR*atoi(*(argv+n));
  else
    if(*(argv+n) == '-')
      switch(*((*(argv+n))+1)){
#ifdef VAX
        case 'c': selfcore = 1; break;
        case 's': liban = 1; break;
        case 'r': nlit = 1;
          n += 1;
          if(n > argc) usage();
          nlin = *(argv+n);
          break;
        default : usage();
      }
    else{
      nlit = 0;
      nlin = *(argv+n);
    }
}

nliz=strlen(nlin);

/*
  Verifications des arguments
*/

if(scons == 0){
#ifdef FOREIGN
  fprintf(stderr, "LE_LISP : no conses, I can't work. 0);
#else
  fprintf(stderr, "LE_LISP : je ne peux travailler sans cons. 0);
#endif
}

```

```

        out();
    }

    if(!liz)
        if(close(open(flin, 0)) != 0){
#ifdef FOREIGN
            fprintf(stderr, "LE_LISP : cannot find file %s0. flin);
#else
            fprintf(stderr, "LE_LISP : je ne trouve pas le fichier %s0.
                                flin);
#endif
        out();
    }

    /*
     * Calcul de la taille de la mémoire
     */

    stbin = scons / 64; /* taille de la table des bits invisibles */

#ifdef TABLGC
    stbgc = (snumb+sfloat+svect+sstrg+ssymb+scons)/32; /* bits GC */
#else
    stbgc = 0;
#endif

    /* en 2 coups a cause des limites de certains compilos C */

    size = sstack+PTR+scode+sheap+svect;
    size = size+snumb+sfloat+sstrg+ssymb+scons+stbin+stbgc;

#ifdef VAX
    if(selfcore == 1){
        corinit(); /* restauration turbo */
        out(); /* au cas ou on rentrerait */
    }
#endif

    /*
     * Allocation de la mémoire
     */

    bmem = sbrk(size);

    if(bmem == RATE){
#ifdef FOREIGN
        fprintf(stderr, "LE_LISP : I can't get required memory space0);
#else
        fprintf(stderr,
            "LE_LISP : impossible d'allouer tant de mémoire0);
#endif
        out();
    }

    estack = bmem;
    mstack = estack + (64 * PTR);
    bstack = estack + sstack;

    bcode = bstack + PTR;

```

```

ccode = bcode;
ecode = bcode + scode;

bheap = ecode;
cheap = bheap;
eheap = bheap + sheap;

bnumb = eheap;
cnumb = bnumb;

bfloat = bnumb + snumb;
cfloat = bfloat;

bvect = bfloat + sfloat;
cvect = bvect;

bstrg = bvect + svect;
cstrg = bstrg;

bsymb = bstrg + ssstrg;
csymb = bsymb;

bcons = bsymb + ssymb;
ccons = bcons;
econs = bcons + scons;

btbin = econs;
btbgc = btbin + stbin;
emem = btbgc;

/*
    et on y va !!!
*/

inton();
llstdio();
llstart();

/*
    au retour (si l'on rentre) on sort joliment
*/

out();
}

```

```
#ifdef VAX
```

```
/*
```

```
    Lancement d'un core BSD4x
```

```
*/
```

```
/* Point de lancement effectif */
```

```
extern llcorgo(); /* dans llvax.sa */
```

```
/* Variables positionnees au moment du save-core BSD4x */
```

```
int corscons; /* taille des CONS de l'image memoire */
int corstbin; /* bits invisibles de l'image memoire */
char *corbtbin;
```

```

/*
  Le lancement
*/
corinit(){
  int diffcons;
  int diffmem;
  int i;
  if(scons > corscons){
    /* l'image memoire est plus petite que lelisp courant */
    diffcons = (scons - corscons);
    diffmem = diffcons + (diffcons / 64);

#ifdef TABLGC
    diffmem += diffcons / 32;
#endif

    if(sbrk(diffmem) == RATE){
#ifdef FOREIGN
      fprintf(stderr, "LE_LISP : not enough resources to allocate space 0);
    #else
      fprintf(stderr, "LE_LISP : impossible de vous fournir tant de place 0);
    #endif

    out();
    }
    econs += diffcons;
    btbin = econs;
    for (i = 0; i < corstbin; i++) btbin[i] = corbtbin[i];

#ifdef TABLGC
    btbgc = btbin + stbin;
#endif
  }
  inton();
  llcorgo();
}

/*
  Save-core BSD
*/

struct exec entete;
char corbuf[1024];

extern int errno;

int corsav (nom) char *nom; {
  int fd, fdbin;
  long symbols;
  long bout;
  int n;

  /* positionne les variables pour le restore-core */
  corscons = scons;
  corbtbin = btbin;
  corstbin = stbin;

  if((fdbin = open(LELISPBIN, 0)) == -1)
    errreturn(LELISPBIN, 1);

  read(fdbin, &entete, sizeof (struct exec));

  if((fd = creat(nom, 511)) == -1){
    if(errno != ETXTBSY)
      errreturn(nom, 1);
  }
}

```

```

        if(unlink(nom) == -1)
            errreturn(nom, 1);
        if((fd = creat(nom, 511)) == -1)
            errreturn(nom, 1);
    }

    symbols = N_SYMOFF(entete);
    bout = (long) sbrk(0);

    entete.a_data = bout - entete.a_text;
    entete.a_bss = 0;

    write(fd, &entete, sizeof(struct exec));
    write(fd, corbuf, N_TXTOFF(entete) - sizeof(struct exec));
    write(fd, 0L, bout);

    lseek(fdbin, symbols, 0);
    while ((n = read(fdbin, corbuf, 1024)) > 0)
        write(fd, corbuf, n);

    close(fdbin);
    close(fd);
    chmod(nom, 493); /* -rwxr-xr-x */
    return(0);
}

```

```

/*
Le restore-core BSD4x est un exec
*/

```

```

char *corargv[5] = { "LE_LISP", "003", "003", "-c", 0};

```

```

int corest (nom) char *nom; {
int fd, i;
    if((fd = open(nom, 0)) == -1)
        errreturn(nom, 1);
    close(fd);
    sprintf(corargv[1], "%d", pipllgo);
    sprintf(corargv[2], "%d", scons/(8*8*1024));
    execve(nom, corargv, envpini);
    errreturn (nom, 1);
}

```

```

#else
/*

```

```

Images memoires
=====

```

L'entete des fichiers image-memoire contient :

- 3 mots d'identification (12 caracteres ascii)
- les tailles des zones de l'image memoire
 - corsstack
 - corscode
 - corsheap
 - corsnumb
 - corsfloat
 - corsvect
 - corsstrg
 - corssymb
 - corscons
 - corstbin
- les tailles des bouts memoires sauves

```

sllm3
ucode
uheap
urest

```

```

*/

```

```

#define ID "lelisp core "

```

```

struct ENTETE {
    char idlelisp[12];

```

```

    int corsstack;
    int corscode;
    int corsheap;
    int corsnumb;
    int corsfloat;
    int corsvect;
    int corsstrg;
    int corssymb;
    int corscons;
    int corstbin;

```

```

    int sllm3;
    int ucode;
    int uheap;
    int urest;

```

```

} entete;

```

```

/*

```

```

l'instruction LLM3 CORSAV doit positionner les 4 variables:

```

```

bllm3, ellm3 limites de la zone impure LLM3

```

```

llucode      fin de la zone code utilisée

```

```

lluheap      fin de la zone heap utilisée

```

```

*/

```

```

char *llucode, *lluheap;

```

```

char *bllm3, *ellm3;

```

```

int corsav (nom) char *nom; {

```

```

int fd;

```

```

    if((fd = creat(nom, 511)) == -1)
        errreturn(nom, 1);

```

```

    strncpy(entete.idlelisp, ID, 12);

```

```

    entete.corsstack = sstack;
    entete.corscode = scode;
    entete.corsheap = sheap;
    entete.corsnumb = snumb;
    entete.corsfloat = sfloat;
    entete.corsvect = svect;
    entete.corsstrg =sstrg;
    entete.corssymb = ssymb;
    entete.corscons = scons;
    entete.corstbin = stbin;

```

```

    entete.sllm3 = ellm3 - bllm3;

```



```

entete.ucode = llucode - bmem;
entete.uheap = lluheap - bheap;
entete.urest = enem - bnumb;

if(write(fd, &entete, sizeof(struct ENTETE))
    != sizeof(struct ENTETE)){
    /* entete */
    close(fd);
    errreturn(nom, 1);
}

if((write(fd, bllm3, entete.sllm3)) != entete.sllm3){
    /* variables LLM3 */
    close(fd);
    errreturn(nom, 1);
}

if((write(fd, bmem, entete.ucode)) != entete.ucode){
    /* code utilisé */
    close(fd);
    errreturn(nom, 1);
}

if((write(fd, bheap, entete.uheap)) != entete.uheap){
    /* heap utilisé */
    close(fd);
    errreturn(nom, 1);
}

if((write(fd, bnumb, entete.urest)) != entete.urest){
    /* zones lisp */
    close(fd);
    errreturn(nom, 1);
}

if((write(fd, btbm, entete.corstbm)) != entete.corstbm){
    /* bits invisibles */
    close(fd);
    errreturn(nom, 1);
}

codereturn(close(fd));
}

int corest (nom) char *nom; {
int fd;
if((fd = open(nom, 0)) == -1)
    errreturn(nom, 1);

if(read(fd, &entete, sizeof(struct ENTETE))
    != sizeof(struct ENTETE)){
    /* lit l'entete */
    close(fd);
    errreturn(nom, 1);
}

if(strcmp(entete.idlelisp, ID, 12)){
    /* chaîne d'identification */
    0, nom);
#ifdef FOREIGN
    fprintf(stderr, "LE_Lisp : %s is not a core image

```

```

#else
    fprintf(stderr,
        "LE_LISP : %s n'est pas une image memoire 0. nom);
#endif
    close(fd);
    errreturn(nom, 1);
}

if((entete.corsstack != sstack)
    || (entete.corscode != scode)
    || (entete.corsheap != sheap)
    || (entete.corsnumb != snumb)
    || (entete.corsfloat != sfloat)
    || (entete.corsvect != svect)
    || (entete.corsstrg !=sstrg)
    || (entete.corssymb != ssymb)){ /* les tailles fixes */
#ifdef FOREIGN
    fprintf(stderr,
        "LE_LISP : non compatible core image : %s 0. nom);
#else
    fprintf(stderr,
        "LE_LISP : image memoire non compatible : %s 0. nom);
#endif
    close(fd);
    errreturn(nom, 1);
}

if(entete.corscons > sconss){ /* trop gros ? */
#ifdef FOREIGN
    fprintf(stderr, "LE_LISP : core image too large : %s 0. nom);
#else
    fprintf(stderr, "LE_LISP : image memoire trop grosse : %s 0. nom);
#endif
    close(fd);
    errreturn(nom, 1);
}

if((read(fd, blm3, entete.silm3)) != entete.silm3){
    close(fd); /* variables LLM3 */
    errreturn(nom, 1);
}

if((read(fd, bmem, entete.ucode)) != entete.ucode){
    close(fd); /* code utilise */
    errreturn(nom, 1);
}

if((read(fd, bheap, entete.uheap)) != entete.uheap){
    close(fd); /* heap utilise */
    errreturn(nom, 1);
}

if((read(fd, bnumb, entete.urest)) != entete.urest){
    close(fd); /* zones lisp */
    errreturn(nom, 1);
}

if((read(fd, btbin, entete.corstbin)) != entete.corstbin){

```

/* bits invisibles */

```

        close(fd);
        errreturn(nom, 1);
    }

    close(fd);
    return(0);
}

#endif

#ifdef S5

int corsav (debut, fin, nom) char *debut, *fin; char *nom; {
    /* a faire sous systeme 5 */
    return(0);
}

int corest (debut, fin, nom) char *debut, *fin; char *nom; {
    /* a faire sous systeme 5 */
    return(0);
}

#endif

/* cline
=====
Avec le pipe magique vers lelispgo pour
eviter de forker toutes les donnees LE_LISP (pour V7).
et pour que ce soit lelispgo qui s'occupe du stty (pour tous).
*/

retcline(){
}

cline(buff)
char *buff:{
    signal(OK, retcline);          /* arme le retour de commande */
    write(pipilgo, buff, strlen(buff)+1); /* passe la commande */
    pause();                      /* et se swappe */
}

/* runtime
=====
Retourne le temps depuis le debut du job.
Ce temps est en secondes (flottant).
L'unité de temps dépend du système
*/

#ifndef TIMEUNIT
#ifdef V7
#define TIMEUNIT 60.
#endif
#ifdef BSD4x
#define TIMEUNIT 60.
#endif
#ifdef S5
#define TIMEUNIT 100.
#endif

```

```

#endif

#ifdef BSD42
double
runtime()
{
    struct rusage urusage;
    struct timeval utimeval;
    getrusage(0, &urusage);
    utimeval = urusage.ru_utime;
    return(utimeval.tv_sec+(utimeval.tv_usec 1000000.));
}
#else
double
runtime()
{
    struct tms timebuffer;
    times(&timebuffer);
    return (timebuffer.tms_utime / TIMEUNIT);
}
#endif

/* sleep
=====
Dort n secondes. Ce temps est en secondes (flottant).
Helas UN*X ne peut dormir qu'un nb de sec fixes
*/

Csleep(f)
double f; {
    unsigned seconds;
    seconds = f;
    sleep(seconds);
}

/* inton intoff
=====
Gestion d'une interruption utilisateur
et d'une erreur de la machine
*/

extern ll_break(), ll_merro(); /* dans llxxx.sa */

inton () {
#ifdef BSD42
    sigsetmask(0);
#endif
#ifdef V7
    alarm(0); /* désarme le sleep */
    signal(SIGALRM, SIG_DFL);
#endif
#ifdef DEBUG
    signal(SIGINT, ll_break);
    signal(SIGILL, ll_merro);
    signal(SIGBUS, ll_merro);
    signal(SIGSEGV, ll_merro);
    signal(SIGFPE, ll_merro);
#endif
}

intoff () {

```

```

#ifdef DEBUG
    signal(SIGINT, SIG_IGN);
#endif
}

/* getenvrn
=====
Recherche d'une variable de l'environnement.
Rempli le buffer donne argument avec la chaine resultat.
Retourne la taille de la chaine.
*/

extern char *getenv();

getenvrn (nom, buff)
char *nom, *buff;{
char *u;
    if(nom = (u = getenv(nom))){
        while(*buff++ = *u++);
        return (u-nom-1);
    }
    return (0);
}

/* getgloba
=====
retourne la valeur associee a un symbole
dans la table des symboles de l'image LE_LISP.
*/

struct nlist elem[2];
int getgloba (strg) char *strg; {
#ifdef BSD4x
    elem[0].n_un.n_name = strg;
    elem[1].n_un.n_name = NULL;
    nlist(LELISPBIN, elem);
    return(elem[0].n_value);
#endif
#ifdef V7
#ifdef Perkin
    strncpy(elem[0].n_name, strg, 8);
    elem[1].n_name[0] = ' ';
#else
    elem[0].n_name = strg;
    elem[1].n_name = "";
#endif
#endif
    nlist(LELISPBIN, elem);
    return(elem[0].n_value);
}

/* et rien pour S5 actuellement */
}

/* la routine de sortie */
out() {
    close(pip1lgo);          /* ferme le pipe avec lelispgo */
    exit(0);                 /* puis sort en beaute */
}

outcore() {
    perror("LE_LISP : restore-core : ");
}

```

```

    out();
}

/* pour eviter un "core-dumped" pour les mauvais signaux */
oupps(n) {
#ifdef FOREIGN
    fprintf(stderr, "LE_LISP : I quit on signal %d0. n);
#else
    fprintf(stderr, "signal %d0. n);
    fprintf(stderr, "OUPPS! J'ai failli faire un core0);
#endif
    out();
}

/* la syntaxe d'appel de LE_LISP sous UN*X */
usage() {
#ifdef FOREIGN
    fprintf(stderr, "Usage : lelisp [-s] [file] [-r file] [number]0);
#else
    fprintf(stderr, "Utilisation : lelisp [-s] [file] [-r file] [number]0);
#endif
    out();
}

/*
Les tests pour callextern
*/

int cchdir (strg) char *strg; {
    return(chdir(strg));
}

char *chome () {
    return(getenv("HOME"));
}

#ifdef BSD42
char *date () {
    char *s;
    s = ctime(time(0));
    s[24] = ' ';
    return(s);
}
#endif

int cmoinsun () {
    return(-1);
}

double ctest (strg,nf,ni,vect) char *strg; double nf; int ni; int *vect;{
int i;
    printf("la chaine est %s0, strg);
    printf("le flottant est %g0, nf);
    printf("l'entier est %d0, ni);
    printf("le vecteur contient vect[0]=%8x,vect[1]=%8x0,
           vect[0], vect[1]);
    i = vect[0]; vect[0] = vect[1]; vect[1] = i;
    return(nf*ni);
}

```

```
cboucle () {
    while (1) sin(3.14);
}
```

7.5 La bibliothèque d'entrée sortie

Voici enfin le module d'entrées-sorties en C que nous utilisons sur tous les UNIX. Nous conseillons l'implémenteur de s'en inspirer grandement pour les instructions LLM3 telles que OPENI, OPENO, INBF, etc.

```

/.....

    Les modules d'interface
    d'entree/sortie
    utilisable sur tous les C.

    Matthieu Devin
    Jerome Chailloux

    Décembre 1984

...../

/*
    Parametres du systeme multi-fichiers
    -----
*/

#ifndef MAXCHAR
#define MAXCHAR 256 /* taille d'un tampon LLM3 */
#endif
#ifndef MAXCHAN
#define MAXCHAN 12 /* nombre de canaux disponibles */
#endif

#ifdef V7
#include <sys/types.h>
#include <sys/stat.h>
#endif

#include <stdio.h>
#include <sgtty.h>

/* le nombre de canaux disponibles */

extern int maxchan;

/* Le flag controlant l'impression des erreurs système */

```

```

/* défini dans lelisp.c */
extern int **prtnsgs;

#ifdef SM90
#define perror(M) printf("%s : system error 0, M)
#endif

#define errreturn(M,V) { if(**prtnsgs != 0) perror(M) ; return(V); }

/* retour des codes d'erreur */

codereturn(v) int v; {
    if(v)
        errreturn("LE_LISP",1);
    return(0);
}

/*
    Les fonctions d'E/S LE_LISP ecrites en C
    =====
*/

/* initialisation du systeme multi-fichier */

llstdio () {
    maxchan = MAXCHAN;
}

/* lecture d'un caractere */

char caractere;
char ttyin () {
    read(0, &caractere, 1);
    return(caractere == '0 ? ' : caractere);
}

/* test de la frappe d'un caractere */

char ttys (buffer) char *buffer; {
    int nchars;
#ifdef BSD4x
    ioctl(0, FIONREAD, &nchars);
#endif
#ifdef S5
    ioctl(0, FIONREAD, &nchars);
#endif
#ifdef V7
    struct stat buf;
    fstat(0,&buf);
    nchars = buf.st_size;
#endif
    if(nchars == 0) return(-1);
    else {
        read(0, buffer, 1);
        if(*buffer == '0') *buffer = ' ';
        return(0);
    }
}

```



```
/* impression d'une chaine de caracteres */
```

```
ttyout (length, buffer) int length; char *buffer; {
    write(1, buffer, length);
}
```

```
/* les buffers d'entree */
```

```
struct {
    char contents[MAXCHAR];
    char *position;
    char *last;
    int filedesc;
} channels[MAXCHAN];
```

```
/* lire sur un fichier
```

Lit la ligne suivante sur le canal argument.
La taille de la ligne lue est rendue dans *ptaille.

Ramène un code condition:

- 0: Ok on a une une ligne complete
- 1: On a rien lu because EOF
- 2: On a lu un debut de ligne qui depasse MAXCHAR
- 3: On a lu la derniere ligne du fichier qui ne se termine pas par CR/LF

```
*/
```

```
int inbf (canal, buffer, ptaille) int canal; char *buffer; int *ptaille; {
    int resread;
    char *ficbuff;
    char *ficpos;
    char *ficlast;
    int ncars;
    ncars = 0;
again:  ficpos = channels[canal].position;
        ficlast = channels[canal].last;
        while ((ncars < MAXCHAR) &&
                (ficpos != ficlast) &&
                ((*buffer++ = *ficpos++) != '0')){
            ncars += 1;
        }
        if (*(buffer-1) == '0'){
            channels[canal].position = ficpos;
            *ptaille = ncars;
            return(0);
        }
        if (ncars == MAXCHAR){
            channels[canal].position = ficpos;
            *ptaille = ncars;
            return(2);
        }
        ficbuff = channels[canal].contents;
        resread = read (channels[canal].filedesc, ficbuff, MAXCHAR);
        if(resread <= 0){
            channels[canal].position = channels[canal].contents;
            channels[canal].last = channels[canal].contents;
            if (ncars > 0){
                *ptaille = ncars;
                return(2);
            }
        }
        else{
```

```

        *ptaille = 0;
        return(1);
    }
}
channels[canal].last = ficbuff + resread;
channels[canal].position = ficbuff;
goto again;
}

/* impression sur fichier

Rend le code condition. */

int outf (canal, length, buff) int canal; int length; char *buff;{
    if(write(channels[canal].fildesc, buff, length) != length)
        errreturn("LE_LISP : outf1 ", 1);
    if(write(channels[canal].fildesc, "0, 1) != 1)
        errreturn("LE_LISP : outf2 ", 1);
    return(0);
}

int outfi (canal, length, buff) int canal; int length; char *buff;{
    if (write(channels[canal].fildesc, buff, length) != length)
        errreturn("LE_LISP : outfi ", 1);
    return(0);
}

/* lseek (uniquement pour des defexterns)
=====
positionnement du canal <chan> a la position <n1>*<n2> */

int lseek (canal, n1, n2) int canal, n1, n2; {
    if (lseek (channels[canal].fildesc, n1*n2) != -1)
        errreturn("LE_LISP : lseek ", 1);
    return(0);
}

/* infile oufile apfile
=====
ouverture d'un fichier en lecture, ecriture, ajout.
Retourne le code condition. */

int infile (canal, buff) int canal; char *buff;{
    if ((channels[canal].fildesc = open(buff, 0)) != -1){
        channels[canal].position = channels[canal].contents;
        channels[canal].last = channels[canal].contents;
        return(0);
    }
    errreturn(buff, 1);
}

int oufile (canal, buff) int canal; char *buff;{
    int fd;
    fd = creat (buff, 511);
    if ((channels[canal].fildesc = fd) != -1){
        channels[canal].position = channels[canal].contents;
        channels[canal].last = channels[canal].contents;
        return(0);
    }
    errreturn(buff, 1);
}

```

```

int apfile (canal, buff) int canal; char *buff;{
int fd;
    if ((fd = open (buff, 1)) == -1)
        fd = creat (buff, 511);
    lseek(fd, 0, 2);
    if ((channels[canal].filedesc = fd) != -1){
        channels[canal].position = channels[canal].contents;
        channels[canal].last = channels[canal].contents;
        return(0);
    }
    errreturn(buff, 1);
}

/* fermeture d'un fichier */

int fclos (canal) int canal;{
    codereturn (close (channels[canal].filedesc));
}

/* detruit, renomme un fichier

Rend un code condition */

int fdele (buff) char *buff;{
    codereturn(unlink(buff));
}

/* mv nom1 nom2 */

int frena (nom1, nom2) char *nom1, *nom2;{
#ifdef BSD42
    if(rename(nom1, nom2) == -1)
        errreturn("LE_LISP : rename ", 1);
#else
    if(close(open(nom1,0)) == 0) {
        unlink(nom2);
        if(link(nom1, nom2) == 0)
            if(unlink(nom1) == 0)
                return(0);
        errreturn(nom1, 1);
    }else
        errreturn(nom1, 1);
#endif
}

```

Table des matières

1 Présentation	
1.1 Démarche à suivre	2
1.2 Matériel nécessaire	4
1.2.1 Fichiers sources	4
1.2.2 Outils	4
1.2.3 Documentation	5
1.3 Temps de portage	5
2 L'expansion	
2.1 Qu'est un macro-expandeur?	6
2.2 Nos macro-expandeurs	6
2.2.1 Le premier expandeur en C	7
2.2.2 Le second expandeur en C	8
2.2.3 Le macro-expandeur <code>LE_LISP</code>	9
2.2.4 Fonctionnement	9
3 Le langage LLM3	
3.1 Quelques unités	11
3.2 Le langage d'assemblage	11
3.3 Description de la machine	13
3.3.1 Les registres	13
3.3.2 La pile	13
3.3.3 Objets manipulés	13
3.3.4 Les types LISP	13
3.3.4.1 Le type <code>CONS</code>	14
3.3.5 Initialisation de la zone	14
3.3.5.1 Le type <code>SYMBOLE</code>	14
3.3.6 Initialisation	15
3.3.6.1 Le type chaîne de caractère (<code>STRG</code>)	15
3.3.6.2 Le type vecteur (<code>VECT</code>)	16
3.3.6.3 Le tas (<code>HEAP</code>)	16
3.3.7 Structure des objets	17
3.3.8 Manipulation indirecte	17
3.3.9 Manipulation directe	18
3.3.9.1 Les nombres flottants (<code>FLOAT</code>)	18
3.3.9.2 Les nombres entiers (<code>NUMB</code>)	19
3.3.9.3 la zone <code>CODE</code>	19
3.4 Espace mémoire	19
3.5 Implémentation	20
3.5.1 Préliminaires	20
3.5.2 Allocation des registres	20
3.5.3 Initialisation de vos registres	21
3.5.4 Accès aux champs des objets LISP	21
3.6 Description des instructions	22
3.7 La bibliothèque (runtime)	22
3.8 Sortie de <code>LE_LISP</code>	22
4 Le lanceur	
4.1 Les variables globales <code>LE_LISP</code>	23
4.1.1 Définition des zones	23
4.1.2 Les limites de la pile	24
4.1.3 Autres variables	24
4.2 Les options de lancement	24
4.2.1 Taille de la zone <code>CONS</code> , et des autres zones	25
4.2.2 Note Importante	26
4.3 Allocation de la mémoire	26
4.4 initialisation de la bibliothèque	26
4.5 Initialisation de la ligne de communication	26

Table des matières

4.6 divers.....	27
4.7 Un exemple de lanceur en C.....	27
5 Le premier interprète	28
5.1 Le lancement.....	29
5.2 Initialisation.....	29
5.2.1 Messages GC_XXXX.....	29
5.2.2 INI_NIL.....	29
5.2.3 INI_TOP.....	29
5.2.4 INI_GC.....	29
5.2.5 INI_PIO.....	30
5.2.6 INI_REA.....	30
5.2.7 INI_PRI.....	30
5.2.8 INI_EVA.....	30
5.2.9 INI_CTL.....	30
5.2.10 INI_CAD.....	30
5.2.11 INI_STD.....	30
5.2.12 INI_NBS.....	30
5.2.13 INI_EXT.....	30
5.2.14 INI_BLL.....	30
5.3 En cas de difficultés.....	31
5.3.1 Aucun message d'initialisation n'est imprimé.....	31
5.3.2 La bannière ne s'imprime pas.....	31
5.3.3 Pas d'écho des caractères lus.....	31
5.4 Chargement de l'environnement standard.....	31
5.5 Test de l'interprète.....	32
6 Le Compilateur	33
6.1 Qu'est le compilateur.....	33
6.2 Comment compiler.....	34
6.3 Ecriture du chargeur.....	34
6.4 Test du Compilateur.....	34
7 Le terminal virtuel	35
7.1 les variables globales.....	35
7.2 Les fonctions.....	36
7.3 Initialisation du terminal virtuel.....	37
7.4 Le lanceur C.....	53
7.5 La bibliothèque d'entrée sortie.....	

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique